Agent Computing Platform for Distributed Satellite Systems

Christopher P. Bridges

Submitted for the Degree of Doctor of Philosophy from the University of Surrey

Surrey Space Centre
Faculty of Electronics and Physical Sciences
University of Surrey
Guildford, Surrey GU2 7XH, UK

August 2009

© Christopher P. Bridges 2009
Abstract

Space and satellite systems are considered to be the most extreme environment to design for and are fraught with engineering difficulty. Performance metrics such as fault tolerance, reliability, pre-determinism and heritage are still high on the list of requirements for all satellite missions. The advent of modern day electronics and miniaturisation, state-of-the-art computing and networking technologies has enabled research into ‘distributed satellite systems’, where multiple spacecraft work collaboratively to perform a mission using intersatellite connectivity. A satellite can be considered one of many nodes in an autonomous and decentralised system, analogous to a mobile ad-hoc network, enabling opportunities in multiple-point sensing, greater communications capabilities, and spacecraft redundancy.

Existing satellite constellations can implement distributed satellite system scenarios but provide unpredictable relative ranges and rates due to various space perturbations. This creates a disconnected environment making it difficult to perform distributed mission operations. Without orbit maintenance, limited onboard resources in power and mass could mean lower processing and networking capabilities which need to rise dramatically to meet requirements for these new missions. This thesis investigates the use of an Agent-based distributed computing platform to enable ad-hoc satellites networking.

Agents for real-time systems and their applications conclude that, despite being utilised in complex control systems, most Agent middleware is unsuited for mission critical, real-time, networked, embedded systems. Two constellation scenarios are simulated for distributed satellite missions highlighting orbital issues such as relative distance and mission lifetime. Computing requirements for such distributed computing opportunities using intersatellite connectivity and Agent technologies have led to a novel system-on-a-chip design, including a general purpose processor core and a dedicated Java co-processing core to enable hard real-time Agent functionalities and software Agent applications at minimal overhead. Common Agent middleware platforms are compared and a software configuration is chosen with relevant Agent services. A distributed image compression case study is also presented. A picosatellite testbed is also designed to provide realistic computing and power constraints.

Keywords: Software Agents, real-time, middleware, distributed satellite systems, system-on-a-chip

Email: c.p.bridges@surrey.ac.uk / cpbridges@blueyonder.co.uk

WWW: www.ee.surrey.ac.uk/SSC
Acknowledgments

I would firstly like to thank and acknowledge my supervisor, Tanya, who has given me many great opportunities to learn and grow as a researcher at Surrey Space Centre. With her belief, her academic and personal support, and her constant and uncanny good advice, I have been able to experience and achieve much at the centre for which I am indebted to. Thank you.

There are numerous students and academics that have given me technical and personal support during my studies. These include, in no particular order, David Barnhart, David Wokes, Nasir Adeli, Dave Bamber, Rudi Weiler, Xiaofeng Wu, Phil Palmer, Karen Collar, Dave Richie, Tom Frame, Bruce Yu, Rafia Mumtaz, Adam Baker, James Valpiani, and Todd Nathaniel. Without all these exceptional friends (and their infinite patience), my understanding of satellite systems, in all its forms, would never have given me such broad knowledge nor the new unending passion for innovation through research I now have.

There are various miscellaneous and what could be described as odd acknowledgements that I have, but acknowledgements none the less. These include tea, a rare pick-me-up in the morning to start and end the day that reminds me of home. House MD and Dexter, to sink myself into and release myself from my studies. For all the ridiculous things I have done, my sanity is (just) in tact.

My parents and family at home have never doubted or stopped me from fulfilling my potential and exploring the world, even if it is only 1 hour away. My mum and dad, my brothers and sister, and all my grandparents have always been intrigued with the stories I tell them. I miss them all dearly. Without all their support and backing, I would never have decided to start such studies and I thank them from the bottom of my heart.

My final and special thanks go to my girlfriend, Kate, who makes my life complete bliss. I could go on about how she’s funny, supportive, and understanding during this process but this thesis would never start. For her love and strength...
# Contents

Abstract ........................................................................................................................................... ii

Acknowledgments ........................................................................................................................ iii

Contents .......................................................................................................................................... iv

List of Figures .................................................................................................................................. viii

List of Tables .................................................................................................................................. xi

Glossary of Terms ........................................................................................................................ xii

1 Introduction ............................................................................................................................... 15

1.1 Motivation ............................................................................................................................. 17

1.2 Scope of Research ............................................................................................................... 17

1.3 Aims and Objectives .......................................................................................................... 18

1.4 Research Novelty ............................................................................................................... 18

1.5 Structure of Thesis ............................................................................................................ 19

2 Distributed Computing Systems .............................................................................................. 21

2.1 Introduction ......................................................................................................................... 21

2.2 Distributed Computing ...................................................................................................... 23

2.3 Distributed Computing in Embedded Systems .................................................................. 25

2.3.1 Embedded Operating Systems .................................................................................... 25

2.3.2 Middleware ..................................................................................................................... 26

2.3.2.1 CORBA ................................................................................................................... 27

2.3.2.2 Java ......................................................................................................................... 28

2.3.3 Real-Time Distributed Computing ............................................................................. 29

2.3.3.1 Real-Time Java for Embedded Network Applications......................................... 31

2.4 Agent Technology ............................................................................................................. 35

2.4.1 Agent Design Methods ............................................................................................... 38

2.4.2 Agent Standards............................................................................................................ 38

2.4.3 Agent Middleware & Open Source Solutions ............................................................ 40

2.4.4 Agents in Embedded Systems ..................................................................................... 41

2.4.4.1 Performance Driven Agent Systems .................................................................... 42

2.4.4.2 Function Driven Agent Systems .......................................................................... 43

2.5 Embedded Networked Systems ......................................................................................... 44
2.5.1 Wireless Sensor Networks ......................................................................................... 45
  2.5.1.1 Wireless Sensor Network Challenges .................................................................. 45
  2.5.1.2 Use of Wireless Sensor Networks in Space ...................................................... 47
2.6 Summary .......................................................................................................................... 48

3 Distributed Satellite Systems ........................................................................................... 50
  3.1 Introduction .................................................................................................................... 50
    3.1.1 Picosatellites ............................................................................................................ 50
    3.1.2 Picosatellite Design Trends ..................................................................................... 59
      3.1.2.1 On Board Computer ............................................................................................. 59
      3.1.2.2 Communications ................................................................................................ 60
      3.1.2.3 Electrical Power System .................................................................................... 61
      3.1.2.4 Attitude and Determination Control System ..................................................... 61
      3.1.2.5 Payload Options ................................................................................................ 61
    3.1.2 Distributed Satellite Missions ..................................................................................... 62
      3.2.1 Current Distributed Satellite Missions .................................................................. 63
      3.2.2 Future Distributed Satellite Mission Challenges .................................................. 65
    3.3 Picosatellite Constellations and Orbital Considerations ............................................. 67
      3.3.1 Orbital Perturbations and Disturbance Torques ................................................... 68
    3.4 Satellite Constellations and Simulations ..................................................................... 70
      3.4.1 Initial Ejection from Launcher ................................................................................. 70
      3.4.2 In-Orbit Constellation Scenarios .......................................................................... 73
        3.4.2.1 String-of-Pearls Constellation Scenario ............................................................. 73
        3.4.2.2 Flower Constellation Scenario ............................................................................. 76
    3.5 Distributed Computing in Space .................................................................................. 79
      3.5.1 Applied Agents for Space Applications ............................................................... 80
      3.5.2 Motivation towards the Agent Paradigm ............................................................... 82
    3.6 Summary ....................................................................................................................... 84

4 Agent Computing Platform .............................................................................................. 85
  4.1 Drivers and Requirements ............................................................................................. 85
  4.2 Design Approach .......................................................................................................... 86
  4.3 Summary ....................................................................................................................... 88

5 Java Co-Processor System-on-a-Chip ............................................................................. 89
    System-on-a-Chip .............................................................................................................. 89
    5.1 Configuration .............................................................................................................. 89
    5.2 Java Co-Processor Memory Considerations .............................................................. 91
Contents

5.2.1 Shared Memory and Caches ........................................................................................................ 92
5.3 Java Co-Processor Design Modifications .................................................................................. 94
  5.3.1 Java Co-Processor Integration and Bus Architecture ........................................................ 96
  5.3.2 Technology Primitives ............................................................................................................ 99
  5.3.3 Hardware Exception Handling ............................................................................................... 99
5.4 SoC Design Simulation Results ............................................................................................... 100
  5.4.1 Logic Utilisation Results ........................................................................................................ 101
  5.4.2 Static Timing Results ............................................................................................................. 102
  5.4.3 Power Estimation Results .................................................................................................... 103
5.5 SoC Design Validation .............................................................................................................. 105
  5.5.1 Bootloader Design and Implementation ................................................................................ 106
5.6 Summary .................................................................................................................................... 109

6 Agent Middleware and Application Development .......................................................................... 111
  6.1 Comparison of Agent Middleware Systems ............................................................................. 111
    6.1.1 Test Methodology ................................................................................................................ 112
    6.1.2 Open Source Agent Middleware Performance Experiments and Results ..................... 113
    6.1.3 Critical Analysis of Existing Agent Middleware ............................................................... 115
    6.1.4 Final Software Stack Configuration .................................................................................... 117
  6.2 Agent Based Service Orientated Architecture ......................................................................... 118
    6.2.1 Instance Manager Thread ................................................................................................ 120
    6.2.2 Agent Mobility Service Overhead and Code Migration .................................................. 122
    6.2.3 Parallel Threading and Agent Behaviours ......................................................................... 125
    6.2.4 Data Distribution using Software Agents .......................................................................... 127
  6.3 Agent Computing Networking Functions ................................................................................ 130
    6.3.1 Topology Reconfiguration ................................................................................................ 130
    6.3.2 Capability Function Definition .......................................................................................... 131
  6.4 Case Study: Distributed Image Compression .......................................................................... 134
  6.5 Summary .................................................................................................................................... 141

7 Agent Computing Platform Implementation on Picosatellite Testbed .......................................... 143
  7.1 ‘S-Cube’ Testbed Design ........................................................................................................... 143
  7.2 Java Co-Processor SoC and Agent Middleware Integration ................................................... 146
    7.2.1 Simulation, Emulation, and Hardware in Loop JOP Optimisations .................................. 147
    7.2.2 Implementation Performance Analysis ............................................................................... 150
  7.3 Test Applications ....................................................................................................................... 153
    7.3.1 RTJS Benchmark Applications ......................................................................................... 153
List of Figures

Figure 2-1. Classification of Current Embedded Networked Systems ........................................... 22
Figure 2-2. Software Layer Model and Implementation Methods .................................................. 22
Figure 2-3. Client-Server Architecture .......................................................................................... 23
Figure 2-4. Agent Architecture Model [12] ................................................................................... 25
Figure 2-5. ACE & TOA Library Footprints (2003 – 2009) ........................................................... 27
Figure 2-6. CDC and CLDC Software Components ..................................................................... 29
Figure 2-7. Real-time Distributed Computing Approaches ............................................................. 30
Figure 2-8. Block Diagram JOP Architecture .................................................................................. 33
Figure 2-9. SHAP Architecture .................................................................................................... 34
Figure 2-10. Block Diagram of the aJ-102 [53] ............................................................................ 35
Figure 2-11. Functional Architecture for P2P Nomadic Agents [72] .............................................. 39
Figure 2-12. a) JADE and b) FIPA-OS GUIs ................................................................................. 40
Figure 2-13. The JADE Wireless Embedded Architecture ............................................................... 41
Figure 2-14. Agent Applications Research Field ........................................................................... 42
Figure 2-15. Sensor Node Architecture .......................................................................................... 45
Figure 2-16. TelosB, BTNode, and Jennic High Power Module Motes undergoing sine vibration tests ......................................................................................................................... 47
Figure 3-1. CubeSat Success Chart ............................................................................................... 51
Figure 3-2. Evolution of OBC Systems (Successes & Failures) ..................................................... 59
Figure 3-3. Picosatellite Mission Comparison ................................................................................. 61
Figure 3-4. An example DSS Scenario .......................................................................................... 62
Figure 3-5. DARPA F6 Program Key Technologies ...................................................................... 66
Figure 3-6. Internal and External Views of PCBSat (Rev. C) [126] ................................................. 67
Figure 3-7. Classical Orbital Elements ........................................................................................... 68
Figure 3-8. Orbital Perturbations Overview ................................................................................... 69
Figure 3-9. Perturbation Affects on Distributed Mission ............................................................... 71
Figure 3-10. Relative Range at 1 cm/s ............................................................................................ 72
Figure 3-11. Relative Range at 5 cm/s ............................................................................................ 72
Figure 3-12. Example String-of-Pearls Constellation ..................................................................... 74
Figure 3-13. Year long simulation on example String-of-Pearls .................................................... 75
Figure 3-14. 3 Month Simulation on example String-of-Pearls ..................................................... 75
Figure 3-15. LEO Flower Constellation ......................................................................................... 77
Figure 3-16. LEO Constellation Range from F1 to All Satellites (4-month) ..................................... 77
Figure 3-17. LEO Constellation Range from F-1 to All Satellites (4-day) .................................... 78
Figure 3-18. Flower Constellation Access times for 9 Picosatellites ............................................ 78
Figure 3-19. Intelligence and Co-ordination Levels for Increasing Agent Use .......................... 81
Figure 3-20. Schematic of information flow in the hierarchical stigmergy task allocation architecture ................................................................. 82
Figure 4-1. Basic Block Diagram of Proposed Solution ................................................................. 87
Figure 5-1. Overview of LEON3 and JOP Design ......................................................................... 90
Figure 5-2. Hardware & Software Layer Design and IPC Methodologies .................................. 91
Figure 5-3. System-on-a-Chip Architecture .................................................................................. 93
Figure 5-4. JOP IP Core Wrapper .................................................................................................. 95
Figure 5-5. Timing Diagram from SimpCon and AMBA2 AHB Interface .................................... 97
Figure 5-6. Behavioural Simulation of Reset, Load and Read from JOP to Memory ................... 98
Figure 5-7. Hardware Exception Reset Loop ................................................................................. 100
Figure 5-8. Routed FPGA Design ............................................................................................... 101
Figure 5-9. Zoomed Power Estimation of LEON3, LEON3 + JOP, and JOP ............................... 104
Figure 5-10. Power Consumption by FPGA Function .................................................................... 104
Figure 5-11. Power Consumption Variance with Temperature ...................................................... 105
Figure 5-12. Pender GR-XC3S-1500 FPGA Development Board ............................................ 105
Figure 5-13. Output from GRMON: a) Startup Code b) IP-core Addresses .................................. 106
Figure 5-14. a) Processors with Separate Memory Systems & b) with SMP systems .................. 107
Figure 5-15. Combination Process for LEON3 and JOP Applications ........................................ 108
Figure 5-16. Post Place & Route Simulation of the SoC Design ..................................................... 109
Figure 6-1. Method Probe for RAM Measurements ....................................................................... 113
Figure 6-2. Comparison of RAM Usage for Platform Startup using Probe .................................... 116
Figure 6-3. Comparison of RAM Usage for Platform Startup using Probe (without FIPA-OS) .... 116
Figure 6-4. CDC and JADE-LEAP Software Configuration ........................................................... 117
Figure 6-5. JADE-FT Middleware and Service Components ....................................................... 119
Figure 6-6. Algorithm for the Instance Manager .......................................................................... 121
Figure 6-7. Instance Manager Classes and Interface Diagram ..................................................... 122
Figure 6-8. Mobile Agent ACL Message Passing for Service Distribution .................................. 123
Figure 6-9. Memory Profile of JADE-FT with Mobile Agent Application ..................................... 124
Figure 6-10. Service Agent Components ....................................................................................... 125
Figure 6-11. Parallel Processing Example ..................................................................................... 126
Figure 6-12. Wireless File Distribution Service - Receiver ............................................................. 127
Figure 6-13. TCP Experiment using FileService .......................................................................... 129
Figure 6-14. UDP Experiment using FileService ........................................................................... 129
Figure 6-15. Overview of the Topology Reconfiguration Scheme ................................................ 130
List of Figures

Figure 6-16. Rule Engine Code Snippets ................................................................. 131
Figure 6-17. Node Separation in Sinalgo a) Initially and b) After 3 Orbits ................. 133
Figure 6-18. Cluster Formation using the CF Algorithm ............................................. 133
Figure 6-19. SSTL DMC Test Image (SSTL DMC-II ©) ............................................... 134
Figure 6-20. Code Snippet for sscSplitImage Service ................................................ 135
Figure 6-21. Image Data Size vs Splitter Service Time .................................................. 136
Figure 6-22. Simulation Diagram for ISL Latencies (Basic Overview) ....................... 136
Figure 6-23. New Round Trip Delay Flowchart ......................................................... 138
Figure 6-24. Round Trip Delay: Single vs N Nodes ...................................................... 139
Figure 6-25. Round Trip Delay: Additional Retransmissions as Variable .................... 140
Figure 6-26. Round Trip Delay: Splitter Service as Variable ....................................... 140
Figure 7-1. Rendered Physical S-Cube Configuration (Front and Isometric Views) ...... 144
Figure 7-2. Demonstrator Satellite Architecture .......................................................... 144
Figure 7-3. ESPACENET Demonstrator System .......................................................... 145
Figure 7-4. Software and Hardware Integration Methodology ....................................... 147
Figure 7-5. Post Place & Route Simulation with Simulated Hardware & Bootloader Load/Operation ................................................... 148
Figure 7-6. jop Verbose Output .................................................................................... 149
Figure 7-7. Footprint Load and Initialisation Confirmation using GRMON ................... 150
Figure 7-8. Post Place & Route Simulation of LEON3 and JOP AHB Bus Requests .......... 151
Figure 7-9. UDP/IP Client Application Class Loading ............................................... 153
Figure 7-10. Normalised Comparison of Micro Benchmark Experiments ....................... 154
Figure 7-11. Simulated JADE-LEAP-pjava Application .............................................. 155
Figure 7-12. ModelSim Stack Error Trace ................................................................. 156
Figure 7-13. Design Changes in JOP Stack and Runtime Changes ............................... 157
Figure 7-14. Experiment Setup for Topology Reconfiguration ...................................... 159
Figure 7-15. PC Console output to demonstrate JADE-FT and Topology Reconfiguration .... 160
Figure 7-16. Instance Manager Thread performing Soft Resets .................................... 161
Figure 8-1. Roadmap to Validating Real-Time Agent Computing .................................. 169
List of Tables

Table 1-1. Spectrum of Satellite Classes, Mass & Costs .................................................. 16
Table 2-1. Footprint Comparison of Embedded Operating Systems .................................. 26
Table 2-2. OS Pre-emption Latency ................................................................................. 31
Table 3-1. Picosatellite Comparison Review ................................................................. 53
Table 3-2. Comparison of Current Distributed Satellite Systems ....................................... 65
Table 3-3. Constellation Mission Characteristics and Applications ................................. 70
Table 3-4. Additional Satellite Simulation Properties ....................................................... 73
Table 3-5. String-of-Pearls Constellation Classical Orbital Elements ............................. 74
Table 3-6. Flower Constellation Orbital Parameters and Phasing for a 9 Picosatellite Flower Constellation ................................................................. 76
Table 3-7. Overview of the Internet Protocol in Space ....................................................... 79
Table 5-1. AHB vs APB Implementation Trade-off ......................................................... 90
Table 5-2. Memory Footprint Comparison ....................................................................... 92
Table 5-3. Device Utilisation Comparison Summary on the Spartan-3 1500 FPGA .......... 102
Table 5-4. Summary of Integration Utilisation on GR-XC3C-1500 FPGA Board ................ 102
Table 5-5. Application Creation Files and Processes for LEON3 RTEMS 4.6 & JOP ........ 107
Table 6-1. Agent Middleware Comparison ...................................................................... 112
Table 6-2. Agent Platforms – Un-optimised Functional Results ....................................... 114
Table 6-3. Classes and Methods that increase the JADE-FT RAM Requirements ............ 124
Table 6-4. Experimental Wireless Parameters and Results of TCP and UDP FileServices ... 128
Table 6-5. Capability Function Encoding Example ............................................................ 132
Table 6-6. Splitter Service Performance Metrics .............................................................. 135
Table 7-1. JOP Implementation Changes ......................................................................... 149
Table 7-2. Bootloader Implementation Method Comparison ........................................... 152
Table 7-3. RTJS Benchmark Applications ........................................................................ 154
Table 7-4. Power Measurements for the LEON3 and LEON3 + JOP Configurations ....... 158
# Glossary of Terms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACE</td>
<td>Adaptive Communication Environment</td>
</tr>
<tr>
<td>A2P</td>
<td>Agent to Peer</td>
</tr>
<tr>
<td>ACL</td>
<td>Agent Communication Language</td>
</tr>
<tr>
<td>ADCS</td>
<td>Attitude Determination and Control System</td>
</tr>
<tr>
<td>AMS</td>
<td>Agent Management Service</td>
</tr>
<tr>
<td>ASIC</td>
<td>Application Specific Integrated Circuit</td>
</tr>
<tr>
<td>AWT</td>
<td>Abstract Window Toolkit</td>
</tr>
<tr>
<td>CDC</td>
<td>Connected Device Configuration</td>
</tr>
<tr>
<td>CDLC</td>
<td>Connected Limited Device Configuration</td>
</tr>
<tr>
<td>CIS</td>
<td>Complex Instruction Set Computer</td>
</tr>
<tr>
<td>CORBA</td>
<td>Common Object Requesting Broker Architecture</td>
</tr>
<tr>
<td>COTS</td>
<td>Commercial Off The Shelf</td>
</tr>
<tr>
<td>CMP</td>
<td>Chip Multi Processor</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>C/S</td>
<td>Client/ Server</td>
</tr>
<tr>
<td>CVM</td>
<td>Classic Virtual Machine</td>
</tr>
<tr>
<td>DF</td>
<td>Directory Facilitator</td>
</tr>
<tr>
<td>DSS</td>
<td>Distributed Satellite System</td>
</tr>
<tr>
<td>DTN</td>
<td>Delay Tolerant Network</td>
</tr>
<tr>
<td>ENS</td>
<td>Embedded Network System</td>
</tr>
<tr>
<td>EPS</td>
<td>Electrical Power System</td>
</tr>
<tr>
<td>FIPA</td>
<td>Federation of Intelligent and Physical Agents</td>
</tr>
<tr>
<td>FIPA-OS</td>
<td>FIPA Open Source</td>
</tr>
<tr>
<td>FPGA</td>
<td>Field Programmable Gate Array</td>
</tr>
<tr>
<td>HTTP</td>
<td>Hyper Text Transport Protocol</td>
</tr>
<tr>
<td>IC</td>
<td>Integrated Circuit</td>
</tr>
<tr>
<td>IIOP</td>
<td>Internet Inter-ORB Protocol</td>
</tr>
<tr>
<td>IP</td>
<td>Internet Protocol</td>
</tr>
<tr>
<td>ISR</td>
<td>Interrupt Service Routine</td>
</tr>
<tr>
<td>JADE</td>
<td>Java Agent Development Environment</td>
</tr>
<tr>
<td>JIT</td>
<td>Just in Time</td>
</tr>
<tr>
<td>JOP</td>
<td>Java Optimised Processor</td>
</tr>
<tr>
<td>JTAG</td>
<td>Joint Test Action Group</td>
</tr>
<tr>
<td>JRE</td>
<td>Java Runtime Environment</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Definition</td>
</tr>
<tr>
<td>--------------</td>
<td>------------</td>
</tr>
<tr>
<td>JRTS</td>
<td>Java Real Time Specification</td>
</tr>
<tr>
<td>JVM</td>
<td>Java Virtual Machine</td>
</tr>
<tr>
<td>KQML</td>
<td>Knowledge Query and Manipulation Language</td>
</tr>
<tr>
<td>KVM</td>
<td>Kernel-based Virtual Machine</td>
</tr>
<tr>
<td>LC</td>
<td>Logic Cell</td>
</tr>
<tr>
<td>LEAP</td>
<td>Lightweight Extensible Agent Platform</td>
</tr>
<tr>
<td>MA</td>
<td>Mobile Agent</td>
</tr>
<tr>
<td>MAS</td>
<td>Multi Agent System</td>
</tr>
<tr>
<td>MANET</td>
<td>Mobile Ad-Hoc Network</td>
</tr>
<tr>
<td>MCU</td>
<td>Micro Controller Unit</td>
</tr>
<tr>
<td>MIDP</td>
<td>Mobile Information Device Profile</td>
</tr>
<tr>
<td>MIPS</td>
<td>Microprocessor (without) Interlocked Pipeline Stages</td>
</tr>
<tr>
<td>MMU</td>
<td>Memory Management Unit</td>
</tr>
<tr>
<td>MPU</td>
<td>Micro Processor Unit</td>
</tr>
<tr>
<td>MTP</td>
<td>Message Transport Protocol</td>
</tr>
<tr>
<td>OBC</td>
<td>On Board Computer</td>
</tr>
<tr>
<td>OMG</td>
<td>Object Management Group</td>
</tr>
<tr>
<td>ORB</td>
<td>Object Request Broker</td>
</tr>
<tr>
<td>OS</td>
<td>Operating System</td>
</tr>
<tr>
<td>P2P</td>
<td>Peer to Peer</td>
</tr>
<tr>
<td>PDA</td>
<td>Personal Digital Assistant</td>
</tr>
<tr>
<td>PIM</td>
<td>Personal Information Management</td>
</tr>
<tr>
<td>PLD</td>
<td>Programmable Logic Device</td>
</tr>
<tr>
<td>RAM</td>
<td>Random Access Memory</td>
</tr>
<tr>
<td>RISC</td>
<td>Reduced Instruction Set Computer</td>
</tr>
<tr>
<td>RMI</td>
<td>Remote Method Invocation</td>
</tr>
<tr>
<td>ROM</td>
<td>Read Only Memory</td>
</tr>
<tr>
<td>RPC</td>
<td>Remote Procedure Call</td>
</tr>
<tr>
<td>RTSJ</td>
<td>Real Time Specification for Java</td>
</tr>
<tr>
<td>SAIC</td>
<td>Science Applications International Corporation</td>
</tr>
<tr>
<td>SEE</td>
<td>Single Event Effect</td>
</tr>
<tr>
<td>SEL</td>
<td>Single Event Latchup</td>
</tr>
<tr>
<td>SEU</td>
<td>Single Event Upset</td>
</tr>
<tr>
<td>SMP</td>
<td>Symmetric Multi-Processing</td>
</tr>
<tr>
<td>SPI</td>
<td>Serial Peripheral Interface</td>
</tr>
<tr>
<td>TAO</td>
<td>The ACE Orb</td>
</tr>
<tr>
<td>TCP</td>
<td>Transmission Control Protocol</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>TMR</td>
<td>Triple Modular Redundancy</td>
</tr>
<tr>
<td>TPTP</td>
<td>Test and Performance Tools Platform</td>
</tr>
<tr>
<td>UART</td>
<td>Universal Asynchronous Receiver/Transmitter</td>
</tr>
<tr>
<td>UDP</td>
<td>User Datagram Protocol</td>
</tr>
<tr>
<td>UI</td>
<td>User Interface</td>
</tr>
<tr>
<td>USB</td>
<td>Universal Serial Bus</td>
</tr>
<tr>
<td>VoIP</td>
<td>Voice over Internet Protocol</td>
</tr>
<tr>
<td>VM</td>
<td>Virtual Machine</td>
</tr>
<tr>
<td>WSN</td>
<td>Wireless Sensor Network</td>
</tr>
<tr>
<td>XML</td>
<td>Extended Mark-up Language</td>
</tr>
</tbody>
</table>
Chapter 1

1 Introduction

This thesis presents the design of an Agent based distributed computing platform utilising ad-hoc networks of satellites, enabling operations for distributed satellite system scenarios. An investigation of current constellations and distributed satellite systems concludes that future distributed satellite missions are unachievable due to strains on resources such as power, computing capability, and propellant for attitude and orbit maintenance.

Since the launch of Sputnik-1 in 1957, humans have designed, built and launched thousands of artificial or ‘man-made’ satellites that orbit around the Earth to enable science, communications, and Earth observation missions. But access to space is an expensive business. The cost of developing systems suitable for the harsh space environment, launch costs and maintaining a ground based control station runs easily into the 10’s of millions of pounds (£). Despite this, the cost of building satellites has being reduced through two significant trends: namely, the use of commercial-off-the-shelf (COTS) parts and miniaturisation.

Surrey Satellite Technology Ltd (SSTL), a spin-out company from the University of Surrey, is the leader in developing and manufacturing small satellites from using COTS components as they are both small in mass and low in cost [1]. These COTS components are tested through strict environment tests such as vibration, thermal vacuum and radiation tests to ensure that they are qualified for space applications. The natural progression of component miniaturisation in modern day electronics has also helped in reducing the physical size and mass of building small satellites so that even smaller satellites can be developed which utilise state-of-the-art integrated electronics solutions, reducing the power consumption also. Table 1-1 shows this variety of satellites that are now in production and highlights linear reduced cost when building very small satellites. All satellites are primarily restrained by the payload instrument with mass reduction made from that initial requirement.

With a new reduced cost for satellite builds, a new class of missions called distributed satellite systems has been the focus in recent research for small satellites [2]. A distributed satellite system will essentially use multiple spacecraft in varying configurations to achieve a mission’s goals collaboratively. These mission concepts require the need for communication between satellites, referred to as intersatellite links (ISL), to build networks of distributed spacecraft for formation
flying missions with the aim of performing distributed operations to increase the science data return per US dollar ($) or Euro (€). E.g. from a communications network or from multiple images or measurements.

Table 1-1. Spectrum of Satellite Classes, Mass & Costs [3]

<table>
<thead>
<tr>
<th>Satellite Class</th>
<th>Mass</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Large Satellite</td>
<td>&gt; 1000 kg</td>
<td>£0.1-1 B</td>
</tr>
<tr>
<td>Medium Satellite</td>
<td>500 – 1000 kg</td>
<td>£20-100 M</td>
</tr>
<tr>
<td>Minisatellite</td>
<td>100 – 500</td>
<td>£5-50 M</td>
</tr>
<tr>
<td>Microsatellite</td>
<td>10 – 100 kg</td>
<td>£1-10 M</td>
</tr>
<tr>
<td>Nanosatellite</td>
<td>1 – 10 kg</td>
<td>£0.1-2 M</td>
</tr>
<tr>
<td>Picosatellite</td>
<td>0.1 – 1 kg</td>
<td>£10-100 K</td>
</tr>
<tr>
<td>Femtosatellite</td>
<td>1 – 100 g</td>
<td>£100-10,000</td>
</tr>
</tbody>
</table>

These missions enter the terrestrial wireless and embedded realms because they too have limited mass, power, communications, and computational ability. However there are specific requirements that are related to the space environment. For example, networking multiple spacecraft could prove difficult due to the significant distances involved, orbital perturbations and disturbance torques that affect the communications of satellites, where if a satellite were to incorrectly point the intersatellite link antenna in the correct direction, the connection is lost or intermittent.

In terrestrial systems, more recent key technologies include distributed computing used in intranets to distribute computationally intensive tasks and wireless network technologies such as those found in laptops for the Internet. Distributed computing is typically enabled by middleware, a software layer offering services to connect software components across a network for integration or sharing computing resources.

Currently distributed computing is applied to two very different areas. One area aims at highly networked, high performance computing for ‘virtual organisations’ with extremely large pools of resources. The other targets embedded environments, such as wireless sensor networks (WSN), where computing ability, memory, power and communications are extremely limited compared to high performance computing systems. New paradigms have been used to realise a distributed computing environment where intelligent Agents is the latest paradigm, where an Agent is a mobile software entity to provide distributed communications or distributed control.
1.1 Motivation

The latest mobile, wireless and embedded devices have led the advent towards accessing computing services through shared infrastructures. Constantly changing environments meant that the typical client-server paradigm, used in traditional networking applications, may be insufficient. ‘Disconnected computing’ (where the network is often lost and found regularly) and mobility are now important issues in wireless and embedded environments where undesirable characteristics such as intermittence, low bandwidth, high latency or high expense are found.

This research proposes the use of software agents for distributed computing applications in wireless processor embedded systems, referring to the evolution of agent-based computing and current state-of-the-art systems. The need for an adaptable distributed computing platform is also highlighted due to the extreme mobility exhibited in recently proposed satellite missions.

Possible applications are identified in the context of the ESPACENET project [4] which is a joint project funded by the British research council, EPSRC, under grant EP/C546318/1. The project aims to develop flexible, reconfigurable, evolvable, and intelligent multi-spacecraft sensing networks for aerospace-based monitoring and diagnostics. Developing such a complex engineering system will directly benefit from technical advancements in ad hoc networking, MEMS devices, low-power electronics, adaptive and reconfigurable hardware, micro-spacecraft, micro-sensors, and etc. It involves four UK Universities - University of Surrey, University of Edinburgh, University of Essex, and University of Kent together with industrial partners such as Surrey Satellite Technology (SSTL), NASA Jet Propulsion Laboratory (JPL), EPSON and Spiral Gateway.

1.2 Scope of Research

To assume energy efficiency, a low resource platform should be chosen with stringent computing and power available. This research, which is part of the ESPACENET project [5], is targeted at a heterogeneous pico-satellite platform for multi-spacecraft networks in low Earth orbit (LEO). As such, the CubeSat platform [6] is to be the primary hardware testbed in this research effort and work should be aimed accordingly.

The use of COTS wireless technologies, such as the IEEE 802.11 standards, means that there is more focus on low power research in physical, middleware, and application layers towards real-time Agent functionalities. Communication protocols found from open-source resources have not been optimised as they are able to be plugged in as services within the Agent middleware easily using Java APIs.
1.3 Aims and Objectives

This research aims at creating a new distributed computing platform for multiple satellite systems with intersatellite communications based on COTS wireless technologies in LEO.

This research is aimed at:

- Advancing the state of the art in space-based real-time distributed computing systems.
- Creating knowledge towards real distributed space applications from terrestrial technologies.

The main objectives of this research are as follows:

- Investigate state of the art terrestrial distributed computing technologies and current distributed satellite missions.
- Review and classify these technologies and highlight computing requirements for missions involving multiple satellites.
- Design and build a low power distributed computing platform that can be used towards a number of meaningful missions.
- Propose a low cost mission scenario that would initially demonstrate the research.
- Validate the work through simulation and experimentation.
- Design and build a low cost satellite testbed to provide a suitable distributed satellite platform and representative requirements.

1.4 Research Novelty

The novelty of this work is:

- The combination of state-of-the-art hardware-based Java processing and software Agent technologies towards a new ‘Agent Computing Platform’ for a hard real-time embedded Agent environment aimed at mobile and complex distributed operations.
- Enabling real-time Java applications as a system-on-a-chip (SoC) design for mission critical systems requiring embedded real-time Java using the LEON3 processor and the Java processor, JOP, with hardware exception handling.
- The creation of a small service-oriented Agent middleware configuration with error management for any software or connectivity problem at low ROM/RAM consumption.
1.5 Structure of Thesis

The structure of the thesis is organised as follows:

Chapter 2 presents the state-of-the-art embedded networked systems in the areas of real-time distributed computing, middleware, Agent technologies, and associated hardware used to enable these distributed applications. The purpose is to identify the key issues in hardware and software towards creating a new computing platform suitable for highly embedded and networked applications. This chapter clearly shows the benefits of combining real-time Java processing with modern existing Agent functionality.

Chapter 3 investigates the current satellite design trends and specifically the more prevalent use of picosatellites. Current and future distributed satellite missions are also discussed with two simulated satellite constellation mission scenarios to illustrate and overcome orbit and perturbation problems. The aim of this chapter is to highlight the use of existing distributed computing in the context of networking in space.

Chapter 4 discusses the previous chapter's technologies and their drivers in distributed computing, satellite design, and distributed space systems. The concept of dividing 'node' and 'network' computing requirements for new distributed satellite systems are also presented to take forward in research for an Agent Computing Platform using a SoC design with the LEON3 and Java co-processor hardware and Agent middleware software.

Chapter 5 primarily focuses on the development, simulation, and test results of a new non-heterogeneous SoC system. Design considerations are introduced for interprocess communication, caches, hardware exceptions, and a shared memory system. Detailed validation through simulation and experimentation is presented with a method for combining bootloaders.

Chapter 6 investigates current Agent middleware platforms and provides a full comparison using a new method probe to log which class and method uses the most memory. After critical analysis, a final software stack is chosen and a service based Agent architecture is explained with new instance management functionality with services for code migration, parallel threading behaviour, and data distribution. New applications are simulated to demonstrate this new architecture include topology reconfiguration and distributed image compression.

Chapter 7 validates the two technologies, Java processor and Agent middleware, implemented on an emulated picosatellite testbed. The picosatellite testbed is used as a representative baseline for providing requirements for the new architecture to meet and a new method for developing real-time Agent computing platforms in both hardware and software is presented. Performance is assessed using testbench applications and power measurements of the new SoC in operation. Demonstration of the Agent platform across a network of differing hardware is also presented.
Chapter 8 concludes this research, clearly identifying the new and novel areas. Future work is also presented here with a road-map to extending this work.
Chapter 2

2 Distributed Computing Systems

This chapter describes the relevant background associated with this research effort. It aims at giving the reader background knowledge in distributed computing systems. After introducing the field in Section 2.1, distributed computing paradigms and middleware are reviewed in Section 2.2. Section 2.3 examines in greater detail the use of distributed computing for embedded systems including operating systems, middleware, and real-time issues. Section 2.4 examines leading Agent systems for distributed computing in this research domain towards new communication and control applications with aims of making savings in power, time, and ease of use. In Section 2.5, wireless sensor networks and mobile ad-hoc networks are investigated along with a review of the ESA Wireless Study carried out under this research.

2.1 Introduction

Quick time to market in the design and manufacture of integrated circuits (ICs) and processors have truly advanced embedded systems technology over the last decade. Many ICs now have a central processor unit (CPU) embedded in them and can handle complex tasks such as networked routing and on-board signal processing. Other advances in hardware speed and memory capacity have also led to additional software layers for greater abstraction.

An embedded system is defined as a specialised computer with hardware and software to perform a particular function often part of a larger system such as a washing machine or digital camera. Embedded networked systems (ENS) can be defined as networks of these embedded systems for distributed applications and resource sharing. The typical embedded networked device, or node, is very small and often handheld, e.g. a mobile phone, and can be introduced in the mobile computing field where nodes are moving around each other. Ubiquitous or pervasive computing, where many ENSs are harnessed in our physical environment, also overlaps with mobile computing terminology [7].

The Nintendo Wii [8] is an example of an ENS where many types of wireless technologies (such as WiFi, Bluetooth, infrared, text messaging, etc) are incorporated for ‘wire free’ gaming. ENSs can be classified into two groups: wired and wireless, where wired systems use direct cable links for both power and communication and wireless systems that use wireless communication
technologies to ease the deployment of ENSs (e.g. a home alarm system). Wireless systems are divided again into static and mobile systems. Static systems have no movement and are the most common networked sensor systems. Mobile systems require greater complexity for power and communication management. This classification is shown in Figure 2-1.

Figure 2-1. Classification of Current Embedded Networked Systems

For this research, only wireless systems are investigated in greater depth.

All of these ENSs require a software stack to run various applications which take advantage of the embedded network environment. The software stack is usually defined using a layered model using the Open Software Interconnect (OSI) Layer scheme [9], as shown in Figure 2-2.

Figure 2-2 shows the common layered software model forming the hardware and software stack often called the distributed computing platform. For any given distributed computing platform, each layer can be defined in either a hardware or software implementation but these layers are a guide and many designs can cross the boundaries of how hardware and software is designed, also shown in Figure 2-2. The term middleware is used to describe the necessary software services
Chapter 2. Distributed Computing Systems

required to connect distributed applications across a network and is often described between the presentation and application layers.

The following sections describe distributed computing and the software stack required, the latest distributed computing paradigm called Agents, and finally the embedded hardware utilised to complete the distributed computing platform.

2.2 Distributed Computing

The aim of this section is to give an overview of distributed computing, associated concepts and technologies used. The most commonly cited definition of a distributed computing system is by A. S. Tanenbaum [10] as a \textit{collection of independent computers that appears to its users as a single coherent system}. Here, distributed computing is defined as a \textit{decentralised and parallel computing, using two or more computers communicating over a network to accomplish a common objective or task}. Distributed computing spans over many sub-fields including \textit{parallel computing} through grids of networked high performance computers as well as \textit{enterprise computing} using the Internet and Intranets [11]. Various communication paradigms are used to implement a distributed computing environment including, client-servers, publish-subscribe, peer-to-peer and more recently software agents.

The \textit{client-server} approach has been the backbone of many different applications in the computing world, from the Internet on desktop PC’s (such as loading a webpage) to wireless connectivity. The client-server network architecture separates the client from the server, shown in Figure 2-3, where the client software can send requests to a server or application server for data.

![Figure 2-3. Client-Server Architecture [12]](image)

Figure 2-3 shows the client-server software components communication through each application. Server software generally, but not always, runs on powerful computers dedicated for exclusive use for running the business application whilst client software generally runs on common PCs or workstations. This has since changed as state-of-the-art electronics have processing and memory resources. Clients can be classified as either fat, thin or hybrid clients where processing and storage can be defined either locally on the client or remotely on the server dependent on client resources and flexibility [13].
The client-server communication has since been updated using different methods such as remote procedure calls (RPC) [14] and Sun Java's remote method invocation (RMI) [15]. RPC and RMI are general terms for interprocess communication technologies to invoke or execute methods in a remote address space. This is often described as basic mobile code and has evolved to mobilise and execute current software Agent methods (described later in this section).

The publish-subscribe paradigm, where one publisher (or server) is able to 'multicast' or send messages to multiple subscribers (or clients). Often part of larger message passing middleware, the publisher posts data on a defined area, often described as a black board, and multiple subscribers can subscribe and get data from the black board at any given time. As there are no timing definitions, the publisher is decoupled from the subscribers meaning that it is more scalable than the client-server paradigm which uses tree based or network address based topology schemes requiring direct communication. This asynchronous operation means that this paradigm provides reliability in often unreliable network situations such as wireless applications. But conversely, this loose coupling often means that messages can be lost as there is no end-to-end confirmation of the message reaching its destination. Another disadvantage is in data security where a publisher could post incorrect or damaging information into a network [16].

The peer-to-peer (P2P) paradigm is where clients and servers are equals, known as peers, where information is decentralised and all users access the resource pool. This means that data is spread across the network and not just on one server where bottlenecks often occur. Predominantly used in ad-hoc networks and the Internet, P2P systems rely heavily on large bandwidth applications for exchanging large amounts of data over a network. P2P systems are being used more in areas where many users need dynamic ad-hoc connections such as the constant connects/disconnects to the Internet for file sharing. The main disadvantage is, as with any open system, that it is vulnerable to malicious attacks [17].

The agent paradigm, described in detail in Section 2.4, is a computational entity that can be viewed as perceiving and acting upon its environment and that is autonomous in that its behaviour at least partially depends on its own experience. Agent code that relocates, including its execution state, on to another processor, to continue execution there is defined as a mobile agent where the code is not communicated as in other paradigms but it transports the code to the server rather than utilising the communication medium as shown in Figure 2-4.
Figure 2.4 refers to the process where a mobile agent can transport its state from one environment to another, with its data intact, and still being able to perform appropriately in the new environment. Mobile agents decide when and where to move next (evolved using RPC) and accomplish this move through data duplication. When a mobile agent decides to move, it saves its own state and transports this saved state to next host and resumes execution from the saved state. However, in contrast to the remote evaluation and code on demand paradigms, mobile agents are active in that they may choose to migrate between computers at any time during their execution. This makes them a powerful tool for implementing distributed applications in a computer network [18].

2.3 Distributed Computing in Embedded Systems

As described in Section 3.1, a common distributed platform is designed using the TCP/IP layer stack system and has more recently been implemented in embedded systems. As the software becomes more complex it typically operates with greater parallelism and will add a great deal of functionality to the user but at the cost of power consumption. This section will consider some of the important factors for embedded software designers: the operating system, middleware and applications.

2.3.1 Embedded Operating Systems

Complex operating systems (OS) have now the opportunity to fit on to embedded microcontrollers or microprocessors with low memory resources. An incorrect decision in OS choice can compromise the designed system and hinder optimisation or implementation. Details such as how an OS protects, instruments, communicates, and handles upgrades are all other factors on which OS is chosen for the overall system. The choice of computing platform must conform to the functionality and application requirements as well as hardware restrictions. The appearance of high-performance, energy efficient, low cost, complex 32-bit processors means that it is not possible to constrain a design path so easily with a small amount of variables (such as
memory or power consumption) and decisions regarding hardware/software interaction is the newest concern for the ENS designer. So as the overall system design gets increasingly more complex, several OS capabilities were created to aid software development and design such as the virtual machine, resource management or scheduler, interrupt nesting, file systems, and the memory management unit [19]. These are described further in Appendix B-1.

One of the most precious resources is the memory. In an embedded system, operating system developers must make their software’s total static memory usage, often called the footprint, as small as possible to take the least amount of ROM or RAM. Table 2-1 shows some of the competing operating systems used in embedded systems.

<table>
<thead>
<tr>
<th>Operating System</th>
<th>Footprint (kB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Windows CE</td>
<td>350 (minimum)</td>
</tr>
<tr>
<td>uCLinux</td>
<td>125 – 256 (+100 for other components)</td>
</tr>
<tr>
<td>QNX</td>
<td>12</td>
</tr>
<tr>
<td>FreeRTOS</td>
<td>200 bytes / 4.4</td>
</tr>
<tr>
<td>SnapGear</td>
<td>Basic: 1M/ IP Stack: 200 (not including drivers)</td>
</tr>
<tr>
<td>RTEMS</td>
<td>Small: 64-128/ Complex: 512K</td>
</tr>
<tr>
<td>TinyOS</td>
<td>3.45 (for complete ad-hoc system)</td>
</tr>
<tr>
<td>Redhat eCos</td>
<td>10 – 100</td>
</tr>
<tr>
<td>VxWorks</td>
<td>Min: 26/ Basic:150/ Full: 250</td>
</tr>
</tbody>
</table>

Open-source and commercial operating systems were investigated and are described in Appendix B-2 showing that, dependent on functionality, a footprint of between 10 kB to 1 MB is most common.

### 2.3.2 Middleware

Previously described in Section 2.2, modern day distributed computing is enabled via middleware which is a software bus with changing levels of abstraction so applications become available to other networked applications and is adaptable towards performance measures and system changes. Defined as “the software layer that lies between the operating system and applications on each system” [24], this emerging technology performs many functions:

- Hiding distribution where applications are made up of many interconnected parts running in distributed locations.
- Hiding locality of the various software components, OS services, and communications protocols that are networked.
• Providing uniform interfaces to developers so applications can be easily written, reused, ported and made to interoperate.
• Supplying common services to general-purpose function in applications to avoid duplicating procedures.

Also briefly described in Section 2.2, there are many types of middleware with message passing middleware being the most common which allows software applications connect to exchange data. Other types include transaction processing monitors, remote procedure call routines, object request brokers (ORB), and enterprise service buses (ESB). Two common implementations that provide middleware services are Common Object Request Broker Architecture (CORBA) [25] and Java technologies [26]. These are discussed in greater depth in the next two sections.

2.3.2.1 CORBA

Common Object Request Broker Architecture is Object Management Group's (OMG) open middleware architecture and infrastructure that supports computer applications across networks. Quoted from the CORBA Website, “Using the standard protocol IIOP, a CORBA-based program from any vendor, on almost any computer, operating system, programming language, and network, can interoperate with a CORBA-based program from the same or another vendor, on almost any other computer, operating system, programming language, and network” [27]. It can be scalable and fault-tolerance to support any heterogeneous hardware system such as a real-time ENS. But conversely, CORBA is often cited as being overly complex with a user manual at over 1100 pages. The library footprints of various implementations are constantly updated online for real-time applications using ACE and TOA ORBs The footprint for both these ORBs are shown in Figure 2-5, to be at 1011 kB and 851 kB respectively.

![Figure 2-5. ACE & TOA Library Footprints (2003 – 2009)](image)

SciSys developed a satellite specific ORB called microORB [28] that utilised Agent technology for control and was flown on SSTL’s UoSAT-12 in 2003 [29], the footprint for their applications.
microORB is an embedded and real-time middleware using the Satellite Onboard Interface Services (SOIS). The SOIS Handbook [30] was created by 10 major space agencies to describe the challenges posed by modern day spacecraft onboard interfaces and details the service architecture of the SOIS services for reference or implementation.

An implementation of CORBA, called OmniORB, was selected for evaluation purposes [31]. A minimum implementation of software components was chosen and included RTEMS OS, C++ IOSTREAMS, TCP/IP stack, 802.11 driver, OmniORB 2 and dynamic library. The footprint of such an implementation amounts to 1.7 MB, very large for an embedded system.

CORBA has since been superseded by Java driven technologies for distributed computing, web services, and Internet applications. The Java Runtime Environment (JRE), as with most software, offers differing revisions dependent on the target platform and only embedded solutions are considered for this research. To run distributed applications, CORBA provides complete end-to-end transport through the session and presentation layer as an object request broker (ORB) whilst JRE utilises RMI. They both provide a software abstraction for potential applications to communicate through utilising stubs of mobile code. Both are considered to be transparent distributed computing paradigms.

2.3.2.2 Java

Since the standard desktop Java runtime environment is far too big for embedded devices, Sun Microsystems provides a version of Java, called Java 2 Micro Edition (or J2ME), which is scaled down to meet the needs of small devices. J2ME is intended to be appropriate for different kinds of embedded devices and divided into two configurations. Configurations are the fundamental components of the runtime environment, composed of a virtual machine and a minimal set of class libraries that enable them to provide the base functionality for a particular range of devices that share similar characteristics, such as network connectivity and memory footprint. Currently, two J2ME configurations are defined: the Connected Limited Device Configuration (CLDC) [32] and the Connected Device Configuration (CDC) [33]. CDC succeeded Personal Java (or pjava), based on an earlier 1.1.8 Java revision. The CDC and CLDC software stacks are shown in Figure 2-6.
Figure 2-6. CDC and CLDC Software Components

CDC is designed for devices that have more memory, faster processors and greater network bandwidth, such as TV set-top boxes, residential gateways, vehicle telemetry systems and high-end PDAs. The CDC includes a full-featured Java virtual machine, and a much larger subset of the J2SE platform (81 classes for the CLDC 1.1 while 305 classes for the CDC 1.0) than the CLDC. As a result, most CDC-targeted devices have 32-bit CPUs and a minimum of 2 MB of memory available for the Java platform and associated applications.

CLDC is the smaller of the two configurations, designed for the devices with intermittent network connections, slow processors and limited memory – devices such as mobile phones, two-way pagers and PDAs. These devices typically have either 16 or 32-bit CPUs, and a minimum of 128 kB to 512 kB of memory available for the Java platform implementation and associated applications.

2.3.3 Real-Time Distributed Computing

The definition of real-time is often misinterpreted as instantaneous but is the completion of a task in a period of time, constrained by the scheduler in the operating system which will expect the task completed in a defined ‘time slice’ of CPU time. Therefore, the performance required of real-time systems is relative to the CPU frequency and timeliness of the associated expected response
time of a particular operation or task. Real-time systems are especially important in situations where important data, expensive components/machinery or (ultimately) human life is jeopardised such as in medical, aerospace, space, and military fields. Real-time software applications are typically defined as being \textit{hard} or \textit{soft} real-time. A hard real-time task must guarantee that applications will meet time deadlines or catastrophic consequences can occur. Soft real-time tasks only aim at minimising missed deadlines so an incorrect system behaviour and any missed deadlines do not cause serious consequences. Hard real-time systems will typically interface at the lowest level of abstraction with hardware such as embedded devices where software is compiled for to specific target. Although this generates the best result, it is not particularly portable to other devices for distributed computing. Soft real-time systems do not have to keep to as strict a timing requirement and, as such, can use higher levels of abstraction such as virtual machines [34].

Hard real-time can be achieved using middleware but also with dedicated hardware. These can be realised as \textit{application specific integrated circuits} (ASICs) or in \textit{reconfigurable computing} technologies, such as the LEON3 processor [35], typically implemented in a \textit{field programmable gate array} (FPGA). FPGAs, unlike, ASICs use reconfigurable logic cells (LCs) and memory blocks to implement various logic to form dedicated functions. Some FPGAs can reconfigure areas of logic cells whilst still in operation which is called \textit{partial runtime configuration} [36]. Fig. 2-6 shows the spectrum of target hardware and software options for real-time distributed computing applications with some examples.

As shown in Figure 2-7, there are many ways of implementing hard-real time based on a particular target device or software configuration. Depending on the choices made, the programming language could be assembly (hard real-time) all the way up to Java (soft real-time).
As described in [37], the operating system and network driver’s delay may be a significant delay source. It discusses that a real-time operating system is required to tackle the pre-emption latency and OS’s were compared in order to decide which will be used for the distributed computing platform (see Table 2-2).

**Table 2-2. OS Pre-emption Latency**

<table>
<thead>
<tr>
<th>OS</th>
<th>Pre-emption Latency (us)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SnapGear (2.6 kernel)</td>
<td>412</td>
</tr>
<tr>
<td>eCos</td>
<td>127</td>
</tr>
<tr>
<td>RTEMS</td>
<td>38</td>
</tr>
</tbody>
</table>

RTEMS has features, which are not available in SnapGear and eCos for example a multiprocessor manager which supports simple and flexible real-time multiprocessing mechanism e.g. for sharing data and providing global resources between many different types of processors.

### 2.3.3.1 Real-Time Java for Embedded Network Applications

In embedded environments, the scenario with very restricted memory, CPU and battery resources is additionally complicated when real time requirements are considered. Normally, considering Java for programming such systems is not ideal but Java has a lot of features that make it very attractive to ENS designers in the mobile computing field, including:

- Heterogeneity with a JVM for Platform Independence
- Security Features and API
- Dynamic Memory Management with Garbage Collectors
- Language is simple and object-orientated
- Suited to Networkability
- Suited to Parallel Operations

When investigating the programming language choice between C, C++, and Java, there are many benchmarks available which show comparisons of compilers, interpreters with varying results from differing benchmarks [38, 39, 40] but C and C++ will be an order magnitude faster than Java. But when working with wireless systems, the communication scheme may be the most latent segment in the whole system which could make the programming language less significant.

There are still many efforts to improve Java’s deterministic behaviour including the use of *just-in-time* (JIT) compilers, the introduction of a real time specification for Java (RTSJ), and various hardware implementations of the JVM.
2.3.3.1.1 Just-in-Time Compilation

Just-in-time compilers convert byte-code to machine code at run time and considered advantageous in many scenarios because it becomes portable, flexible and greatly improves performance. This is called *dynamic classloading* leading to a constantly changing Java environment. The major two drawbacks are 1) the start-up time delay which can be significantly higher and 2) an unpredictable memory model. An embedded JIT compilation method is discussed in [41] on a Yari processor that uses 1 MB RAM but also comments on the large Java classes required which must be statically (precompiled) or dynamically loaded (runtime compilation). An even smaller virtual machine is written in C called KVM [42] and utilises only 80 kB, much like an embedded OS, but is not open-source and has limited functionality.

2.3.3.1.2 Real Time Specification for Java

As Java is such a popular language to use, the Real-Time for Java Expert Group (or RTJEG) made a real-time specification for Java (RTJS) to address timing constraints in Java [43]. The main enhanced areas this specification adds includes:

1. Thread Scheduling and Dispatching
2. Memory Management
3. Synchronisation and Resource Sharing
4. Asynchronous Event Handling (similar to hardware interrupts)
5. Asynchronous Transfer of Control (time-bounding context switching for complex programs)
6. Asynchronous Real-time Thread Termination (an ‘oracle’ to manage real time threads activity).
7. Physical Memory Access (direct access to raw or physical memory areas)

These additions allow programmers to include real time multi-threaded operation and a new long and short term memory management scheme called *scoped memory*. There is also a handler to perform context switching between tasks to guarantee real-time slot completion.

A comparison of ORBexpress middleware versions were tested, written as ‘C++ ORB’ and ‘Java ORB’ [44]. Timing experiments were performed where Java program speeds very comparable to C++. Overall, it summarised that Java optimisations are very necessary but even simple tests of the each C++ and Java implementations resulted in very comparative results. An example experiment was transporting data packets of 32 kB through sockets where the ORB overhead is 81 us in C++ and 85 us in Java.
The latest Java real-time specification is used in [45] for memory management and real-time applications using ‘active’ software components that hide coding complexities and allows them to be reusable blocks of code. It did however assume applications are scheduled offline and parameters and mechanisms for handling overruns and missed deadlines are ignored, part of their future work. An example of using Java in embedded systems is also proposed in [46] where Java is used to reduce hardware communications overheads on an FPGA (tested in Simics) but does not address any real-time issues.

2.3.3.1.3 Hardware Implementations of the JVM

The most recent hardware solutions in this research use early Java revisions to implement deterministic Java execution processors. Two open source Java processor options based on VHDL IP cores are Java Optimized Processor (JOP) [47, 48] and Secure Hardware Agent Platform (SHAP) [49]. JOP, shown in Figure 2-8, was designed by M. Schoeberl in 2003 and implements a JVM in hardware with predictable execution time for embedded real-time systems. It is a RISC architecture with 4 pipeline stages which can be used to predict the number of clock cycles and execution time for Java bytecodes to be run. Results in [50] showed that JOP is the smallest hardware realization of the JVM available to date in between 1100 to 1800 LCs (configuration dependent). Implemented in an FPGA, JOP also has the highest known operating clock frequency of Java processors, operating at 100MHz (limited only by a selection of target FPGA devices). JOP was also compared against several embedded Java systems and, as a reference, with Java on a standard PC to find that a Java processor in hardware is up to 500 times faster than an interpreting JVM in software on a standard processor for an embedded system. This system requires additional software for networking, garbage collection and scheduling.

![Figure 2-8. Block Diagram JOP Architecture](image)

33
JOP research has been extended in [51] for chip-multiprocessor (CMP) use where 3 test-bench applications are tested using between 1 and 8 JOP cores. Their results show that the parallel architecture is faster for multiple threaded behaviours compared to other Java processors but disadvantages include a) the complexity in software design, b) saturation if more tasks are added, and c) loss of processor speed due to processor bus arbitration.

SHAP, shown in Figure 2-9, is another VHDL IP core for running real-time Java but with some improvements over JOP. It is still a RISC architecture with 4 pipeline stages and there is additional hardware support for a schedule, general garbage collection, real-time garbage collection and dynamic class loading instead of software. This functionality though increases the required logic cells and the implementation can be between 50 – 100% more costly compared to JOP. To date, a CMP version of SHAP has not been published. The research aims at eventually running software Agents (1 per core) but no existing Agent software or any new Agent definitions have been published.

![SHAP Architecture](image)

Figure 2-9. SHAP Architecture

Other existing processors include various aJile processors, such as the aJ-100 [52], a one-chip microprocessor to directly execute JVM bytecodes and supports J2ME CDC/CLDC stack and the RTJS. Their next generation processor due at the end of June 2009, the aJ-102 [53], focuses directly on highly embedded and networked applications with added 10/100 Ethernet core, encryption/decryption core, and AMBA AHB interface, shown in Figure 2-10.

It is worth noting that the new aJ-102 architecture follows a similar approach to the one proposed in Chapter 4 of this thesis which was first published in March 2008 [54] and further published in [55, 56]. This architecture is subtly different to the one in this thesis as it does not have the LEON3 processor and has an increased ROM and cache size.
Other implementations include the Imsys IM1101 Java microprocessor [57] and the Javalin Stamp Module [58], both aimed at developing highly networked and embedded Java systems.

The advent of highly integrated circuits and networking applications has obviously led to the development of Java based processors making application development in gaming, mobile phone and many other industries inherently simple, portable, multi-threaded and distributed. As JOP is both open source and the fastest hardware implementation of a JVM, this core could be taken forward for development.

### 2.4 Agent Technology

In computer science, the term software agent is defined in many ways: as an abstraction, an idea, a concept (similar to object-orientated programming using terms such as methods, functions and objects). But unlike objects, described in terms of methods and attributes, agents are described in terms of its behaviour. An agent is essentially a logical model that describes software conveniently to describe powerful and complex software. The main idea is that agents are not strictly invoked for tasks but activate themselves autonomously on behalf of a user. This heavily contradicts with offering deterministic behaviours so any implementation of Agent technology cannot be defined as real-time. Only specific technological or functional implementations can be deterministic. Previous related research fields include distributed artificial intelligence (DAI) and distributed problem solving (DPS), but have now evolved into Agent technology.

There is no unanimously agreed definition of an Agent but G. Weiss [59] gives the best definition where an Agent is “a computational entity that can be viewed as perceiving and acting upon its environment and that is autonomous in that its behaviour at least partially depends on its own experience”.

---

**Figure 2-10. Block Diagram of the aJ-102 [53]**

![Block Diagram of the aJ-102](image-url)
From this point on, G. Weiss’s definition of an Agent is described with a capital ‘A’ to prevent confusion with other definitions of the word ‘agent’.

An Agent is another kind of software abstraction, just like methods, functions and objects in C++ programming. An object is a high-level abstraction that describes methods and attributes of a software component. An Agent, however, is an even higher software abstraction to provide a convenient and powerful way to describe complex software entities. Rather than being defined in terms of methods and attributes, an Agent is defined in terms of its behaviour. This can be sequential or parallel behaviours, one-time or repeating behaviours, proactive or reactive behaviours, or a mix of these behaviours. It is important to note that programming an Agent-based system is primarily a matter of specifying Agent behaviour instead of identifying classes, methods and attributes. Unlike other code, an Agent is capable of operating alone without user intervention, communicating with other available Agents, and processes in the middleware environment. Behaviours can achieve specific software service functions which can be in real-time for disconnected computing environments [60].

Nwana and Ndumu [61] believe that little technological progress has been made since 1994 where researchers have been ‘reinventing the wheel’ and avoided the real issues (distribution problems in multi-agent systems and application, security, management and scalability in mobile agent systems). This paper does however point out the main research topics that the field is lacking in.

Hector proposes the latest classification of software agents [63] that spans all fields of research in 6 categories: pro-activeness, adaptiveness, mobility, collaboration, veracity and disposition. Many of the definitions inherently crossover (e.g. a ‘truthful Agent’ is inherently positioned towards collaborating with other Agents) but provide a classification to help analyse Agent functions. Hector does however say that the classification is not definitive and refining research work is required. Related and derived concepts include many types of Agents. But we are interested in the literature surrounding two types: multi Agent systems (MAS) and mobile Agents (MA).

Multi Agent systems (MAS): When several agents interact together they may form a multi-Agent system or multiple Agent system (also described as an agency). Characteristically such Agents will not have the required data or methods to achieve an objective and must collaborate with other agents to achieve their goal. These systems often have little or no global control and are sometimes referred to as swarm systems where data is decentralised and execution is asynchronous. A design methodology for multi Agent systems is presented in [64] defining Agent rules, behaviours, resources, and user interactions.

Mobile Agents: The next level from object-orientated programming, software agents are the ideal solution for modelling and implementing complex adaptive systems (CAS), especially since the Agent metaphor itself was influenced from CAS modelling. Software agents provide a much more
appropriate metaphor for exploiting the parallelism and dynamics of large-scale computer networks and distributed systems. Asynchronous or autonomous operation is natural for independent software agents, as opposed to the clumsiness of synchronized procedure calls (even if object-oriented) between networked computers. Some main advantages which mobile Agents have over conventional Agents include [65]:

- Mobilise computations to the data source, reducing network load.
- Asynchronous execution on multiple heterogeneous network hosts (i.e. the Agent works even if the hosts are not completely identical).
- Dynamic Adaptation: The Agent bases actions for a move dependent on the state of the host environment.
- Tolerant to network faults: They can operate without an active connection between client and server.
- Flexible maintenance: This means to change an Agent's actions, only the source (rather than the computation hosts) must be updated.

The differing paradigms have been thoroughly compared by A. Puliafito [66]. This paper evaluates each paradigm using Petri nets and presents results where the client-server paradigm is best if the same servers are constantly used, a performance characteristic that is soon outperformed in larger networks. Remote evaluation and mobile Agents outperform the client-server method when 30% of the servers could be different to the previous data communication request. At data rates of 10 kbps, remote evaluation is the best method for time completion vs. throughput. The mobile Agent paradigm outperforms the remote evaluation method when comparing network speeds. At high speeds of 1 Mbps, mobile Agents can utilise all the bandwidth but at slower speeds (less than 100 kbps). Remote evaluation is the best method because a lower amount of data is required for code migration than the mobile Agent paradigm. Mobile Agents therefore need to be carefully selected with several factors in mind (such as program size or network speed). Applications for mobile Agents include resource availability, discovery, monitoring, information retrieval, network management and dynamic software deployment.

Some other advantages that agents have include [67]:

- Mobile agents solve the client/server network bandwidth problem. By moving a query or transaction from the client to the server, the repetitive request/response handshake is eliminated.
- Agents reduce design risk by allowing decisions about the location of code (client vs. server) to be pushed toward the end of the development effort when more is known about
how the application will perform. The architecture even allows for changes after the system is built and in operation.

- Agent architectures also solve the problems created by intermittent or unreliable network connections. Agents can be built quite easily that work “off-line” and communicate their results back when the application is “on-line”.

### 2.4.1 Agent Design Methods

Agents are designed in three methods, described in greater detail by Bordini in [68]:

- **Declarative designs** are formal and utilise base logic (usually in maths/calculus styles)

- **Imperative designs** are made using non-Agent orientated methods (usually from ‘legacy’ programmers in C or C++ as Java is the most popular choice for Agent designing)

- **Hybrid designs** use both declarative and imperative design approaches where Agents can be described declaratively yet with some specific constructs in imperative languages

Bordini also notes observations on Agent designing trends where some designs are from scratch, directly encoding some theory of an agency, while others extend existing languages to suit the Agent paradigm. Using these languages, instead of more conventional ones, proves useful when the problem is modelled as a multi-Agent system, and understood in terms of cognitive and social concepts such as beliefs, goals, plans, roles, and norms.

Current Agent frameworks are not tightly coupled with one specific programming language. Instead, they are concerned with providing general techniques for specific aspects such as Agent communication or coordination. Most mature languages are accompanied by some Integrated Development Environment (IDE) to enhance the productivity of programmers by automating tedious coding tasks (project management, creating and editing source files, build and run process). But the field is overrun with custom software with no real compliance to one type of design methodology. This in itself is a hindrance, where Agent systems take long to develop, and an advantage, making customisable code for certain specifications.

Java is, by far, the most common programming language for programming Agents, with few implementations designed for embedded systems. This means that a JRE is needed to run these middleware systems.

### 2.4.2 Agent Standards

The Foundation for Intelligent Physical Agents (FIPA) formed in 1996 to produce software standards for heterogeneous and interacting Agents and Agent-based systems [69]. Despite having
only 33 active members they are working on many interesting standards [70]. They have specifications spanning Agent applications, communications (Knowledge Query and Manipulation Language (KQML) protocols, acts and languages), management and message transporting (envelopes and transport protocols).

FIPA are currently developing a ‘Peer-to-Peer Nomadic Agents Reference Architecture’ which is to be used to support distributed computing on small or embedded devices. Some problems they are addressing are Agent-to-Peer (A2P) architectures and generic Agent services. Figure 2-11 depicts the latest FIPA architecture for embedded systems with three layers: Generic Agent Services, the Agent platform (including platform services and Agent to peer interface) and P2P Core and Services. This was left as a draft document and the workgroup was closed 2007 [71].

![Diagram](image)

**Figure 2-11. Functional Architecture for P2P Nomadic Agents [72]**

As demonstrated in FIPA’s Nomadic Application Support Specification [72], many communicative acts are defined to support disconnections, mobility and reliable connectivity with any ontology. Many types of communicative acts are available describing 1) which agents are involved, 2) the message content, 3) a message description and 4) who controls the conversation using an Agent Communication Language (ACL) from the Message Transport Protocol (MTP) specification [73]. Using this method, all messages can communicate, regardless of platform. If a message is not received properly, a standard ‘not-understood’ message can be replied to the sender asking to resend the information (if necessary).

AgentLink is Europe’s Coordination Action for Agent-based computing. It coordinates research and development activities in the area of Agent-based computer systems on the behalf of the European Commission. AgentLink supports a range of activities aimed at raising the profile, quality, and industrial relevance of Agent systems research and development in Europe, and promoting awareness and adoption of Agent technologies. Importantly, they developed the AgentLink Roadmap which suggests how Agent based computing could develop over the next decade [74].
2.4.3 Agent Middleware & Open Source Solutions

Most commonly used middleware for Agent design is either CORBA or Java’s Runtime Environment, described in Section 2.3.2. As with most software, they both offer differing revisions dependent on the target environment. A comparison of Java’s RMI and CORBA can be viewed in [75] and [76], but the applicability of one technology over the other still depends on:

- Application (Performance and memory constraints being the largest factors)
- Programmer’s experience
- Maintenance of the distributed system
- Whether non-Java systems are intended to access the system now or in the future.

Two development platforms have been widely adopted for development: Java Agent DEvelopment (JADE) [77] and FIPA-OS [78]. Figure 2-12 shows both Agent graphical interfaces provided by both a) JADE and b) FIPA-OS.

JADE is the most commonly used open-source Agent middleware framework in compliance with the FIPA specifications for interoperable intelligent multi-Agent systems. JADE is very widely used in the teaching, academic and industrial communities and up to date with any FIPA standards and changes for real applications. The goal is to simplify the development while ensuring standard compliance through a comprehensive set of system services and Agents. It deals with all programming aspects that are independent of the applications, such as behaviours, message transport, encoding and parsing, or Agent life-cycle. Fabio Bellifemine, head of the JADE project, is heavily involved with FIPA and chairs their FIPA Architecture Board. As such, JADE, is very widely used in the teaching, academic and industrial communities and up to date with any FIPA standards and changes.
Chapter 2. Distributed Computing Systems

There is also Java Agent Development framework and the Light Extensible Agent Platform (JADE-LEAP) which is a JADE based middleware for embedded devices and other resources constrained devices using wireless links to develop Agent systems and novel application areas such as web or service orientated computing [77], shown in Figure 2-13.

Figure 2-13. The JADE Wireless Embedded Architecture

FIPA-OS (open-source, not operating system) provides examples of containers, management systems and communication services that follow the FIPA specification to extend in your own applications.

Both platforms provide a management services for collaborative development/ co-ordination on a base platform for interoperability and testing purposes. Reduced platforms for embedded devices have been including JADE-LEAP following FIPA’s Nomadic Application Specification.

2.4.4 Agents in Embedded Systems

Agents offer the means to communicate and operate in new and novel ways. This, and associated technology developments, has enabled Agent research to cover many fields. The latest terrestrial distributed and networked systems are now using Agent systems to cope with large scale remotely located services and systems [79]. Relevant Agents systems are included in groundstations to aid in image signal processing [80] and on-board satellites for in-situ autonomous behaviours [81].

Agents are a higher abstraction of programming to deal with complex computing problems. By executing behaviourally and assigning an agent a ‘role’, communication interactions and autonomous actions become easier to realize. This allows them to work proactively and reactively to their environment and to any given task. They can be proactive when finding new communications routes in a networked environment and reactive to disconnections, low bandwidths or high latencies [82].

The current state-of-the-art Agent research is disseminated in different fields dependent on the application area. Here, the research is divided on either the target platform in hardware or the
Agent functionality in software. Figure 2-14 shows the streams of Agent research and some of the performance or functional drivers.

<table>
<thead>
<tr>
<th>Hardware</th>
<th>Software</th>
</tr>
</thead>
<tbody>
<tr>
<td>Embedded Devices</td>
<td>Middleware &amp; Distributed Computing</td>
</tr>
<tr>
<td>High Performance Computing</td>
<td>Web Services</td>
</tr>
<tr>
<td>Social &amp; Network Simulation</td>
<td></td>
</tr>
</tbody>
</table>

**Performance Drivers:**
- Hard/ Soft Real-time
- Low Memory
- Real Implementations

**Functional Drivers:**
- Communications
- Control

Figure 2-14 points out that Agent systems, just like other distributed computing applications, have real-time considerations for both the target hardware and required software stack. The application domains are also very different, from operating on a mobile phone to being abstract software objects in social simulation. Performance driven Agents are discussed along with functionally driven Agent systems, broken down into communication and distributed control purposes.

### 2.4.4.1 Performance Driven Agent Systems

When implementing Agents, they can be defined in hardware logic or as software programs. From the loose definition of Agent terminology, they could potentially be described in hardware description languages (HDL) in LCs of an FPGA. Hardware Agents in [83] by T. Hindam are described as dedicated processors (1 processor per Agent) to provide synchronicity, serve incoming requests to supply local sharable data, and serve internal threads with data from non-internal hardware Agents. Although there are no results presented for this model, it recognizes the future work with this hardware agent with intercluster prefetching, major research topics in system-on-a-chip and other multi-core systems. In [84] by M. G. Panteleyev, a Rete algorithm is used to develop a custom processor based on a set of Agent-based rules so that the processor and data can be implemented on a PLD for production systems. This method aims to increase performance by 20% compared with general purpose processors. Even through this design is highly custom, the Agents are still developed initially in software which requires the knowledge of all nodes in a fixed networked to make savings in memory register usage. If implemented on an FPGA, this system could potentially become reconfigurable. A self-reconfigurable agency is proposed in [85] by Y. Meng. Here, a Virtex-2 FPGA is programmed using C++ to create an
Agency and manage sub-sections of the device to change Agent roles. When reconfiguring in this way, a potential problem would be how to guarantee Agent’s state persistence and memory coherence.

In software, you also have real Agent program implementations for real-time applications or Agents for simulation purposes. For these implementations, there are many ways to implement functionality and layered model such as operating systems or middleware are often included. Notably, K.J. Lin and C. S. Peng in [86] investigate different scheduling algorithms for soft guarantees. The CoBERT Agent [87] offers a similar software route for real-time scheduling and functionality towards simulation. Meta-controls at various time rates are discussed to simulate real-time at node level (in ms) or making and adapting behaviours over a network (in several minutes). Another simulation of an Agent based software model is presented by L. C. DiPippo in [88] using an operating system and CORBA ORB with an Agent Communication layer. An extension of KQML is extended to include real-time constraints (albeit measured in seconds). No implementation results of this system were presented.

2.4.4.2 Function Driven Agent Systems

The embedded wireless community that use multi-agent systems, all concentrate on improving robustness and reliability in mostly the communications realm, where most of the power consumption is used in embedded devices. But trends in Agent use are primarily in:

- Building blocks for control systems to parallelise a device’s computing functions
- Communication techniques and protocols for improved reliability and robustness

An example of using Agents in complex control is to disseminate between differing features or customer requirements from raw data in satellite imagery in a paper by C. Toomey [89]. Agents systems are broken down from an Agency to the Agent implementation using a wrapped C code with tcl scripts and KQML for communication. This is a simple brokering scheme where Agents are static but effectively used to automate customer orders in massive data sets. Brokering was again used by NASA in 2004 for planetary exploration [90], where an astronaut gave voice commands to Agents that then perform tasks and control rovers autonomously whilst allowing feedback from the rovers and health monitoring equipment in the astronauts suit. Another control system is described by K. K. T. Thanapalan in [91], where Agents are used for event based autonomy and sensory data fusion of local instruments. The fused position and attitude data is also distributed to other satellites. No results are shown on the software performance of JACK for this application. K. Wang [92] describes an Agent as a singular node in a constellation for implementing control laws. This, again, highlights the discrepancy in the Agent definition.
Using Agents for communication applications include Smart Messages (SMs), presented by P. Kang [93], where a Java based portable middleware is used for ad-hoc networking, routing and migration. They share the idea of code migration with mobile Agents but unlike Agents, SMs are responsible for their own routing in a network. This feature allows SMs to adapt quickly to changes that may occur in network topology and node resources. A mobile Agent usually has a fixed address and often knows the network configuration, while an SM names nodes by content and discovers the network. A ‘mailbox’ scheme is also presented by J. Cao [94] in a two-tiered fault tolerant message passing scheme. Simulation results are provided that shows their fault tolerant architecture can effectively handle both network and host failures while keeping the communication cost low. An improvement to JADE-LEAP’s communication subsystem in described by A. Macías-Estrada [95] for multicasting messages, where ‘ghost’ Agents are used as an address to various other agents on differing platforms. This involves the starting, managing (registering and deregistering) and destroying of various ghost Agents and their containers to map addresses from low level components to high level components. This work again concentrates on a more robust communications protocol to save energy otherwise spent retransmitting lost messages. An initial basis for a satellite specific framework is described using Agents systems which was investigated by the Science Applications International Corporation (SAIC) for NASA and, specifically, evolution in an Agent community in [96] by C. Rouff. An ontology is defined here but only asks questions with regards to resource management. They argue that unlike a typical distributed computing platform that typically has fixed network characteristics, Agents could be employed to overcome and discover their given network situation.

2.5 Embedded Networked Systems

There are many types of embedded hardware to implement ENSs. These typically include microprocessors (MPUs), microcontrollers (MCUs), as well as ASICs and FPGAs, described in Section 2.3.3, as well as additional networking devices such as radios and antennas. There are two major differences between MPUs and MCUs: instruction set and inputs/output (I/O) resources. MPUs typically have a large instruction set for performing many functions making them flexible in functionality whilst MCUs have a dedicated small instruction set for set peripherals or I/O interfaces. MCUs however have incorporated interfaces and memory into 1 device whilst MPUs still need other devices to support the memory and interfacing that often increases power consumption. As the MCUs are normally much smaller, the often draw less power than MPUs, ASICs, or FPGA designs. Work however by D. Elléouet [97] details the consumption of a Xilinx Virtex 4 FPGA with the LEON processor with a power consumption of around 20 mW, similar to micro-controllers units (MCUs) and micro-processors units (MPUs).
Many of these low power devices are used in ENSs towards wireless sensor networks (WSNs) which are collections of ENSs that typically operate sensors that collect data and route it to a base station, often referred to as the sink [98]. Further to this, a mobile ad-hoc network (MANET) is a WSN in a mobile environment so that implementing routing and communications becomes increasingly complex [99].

2.5.1 Wireless Sensor Networks

As described in Section 2.5, a wireless sensor network is a network of sensor nodes and a base station, which collects data from all the sensor nodes. K. Römer [100] investigates the many WSNs applications such as prototypes in military area monitoring (to detect troop or vehicle movement), crop growth, and stock management but are envisioned be used to make robot skins, smart floors, high energy particle detection, and space exploration [101, 102, 103].

The standard WSN hardware kit, often called a mote, consist of 4 main subsystems: power unit, processing unit (an embedded system of sorts), sensing unit including sensing circuitry, and communications unit typically radio transceivers as found in Figure 2-15.

![Figure 2-15. Sensor Node Architecture](104)

These nodes are classically bought in sets or kits. Some typical wireless sensor kits include the TelosB [105], BTNode [106] and Jennic [107] motes, also used in a study for WSNs for space applications [108]. Appendix A summarises some of individual WSN nodes features. WSN kits use multiple technologies from varying MCU and processing abilities with software such as ZigBee mesh networking and Bluetooth connectivity.

2.5.1.1 Wireless Sensor Network Challenges

With strict requirements in functionality, cost, volume and power consumption, the hardware platform must have a suitable hardware architecture and memory to carry out the software functionality whilst consuming the least power. Processors such as x86, MIPS, PowerPC, Hitachi SH, SPARC, and Strong Arm processors are usually chosen for such embedded environments [109]. Recent families such as the MSP430 series of MPUs have the capability to turn off internal
peripherals to conserve power making them very low power [110]. Often, a low active state in a
given time period or duty cycle is used to reduce operational power but the WSN design must
make a power/ performance and cost trade off [111].

Most of these devices are manufactured as ASICs but the use of open source processor designs to
be used in FPGAs is becoming commonplace in industry and research institutions as a cost
effective solution to various embedded needs [112]. Open source processors include the LEON3
[35] or the ERC32 [113] with numerous interfacing possibilities that can be implemented on
smaller FPGAs such as the Spartan-3 [114] or the larger Virtex-5 FPGAs [115]. This technology
also leverages towards system-on-a-chip (SoC) or network-on-a-chip (NoC) architectures, where
many IP cores are utilised in a device with a network or bus, for growing data-intensive
applications by designing using replication, rather than custom design [116]. Z. Stamenković
[117] presents an ASIC SoC based sensor node using the LEON2 as a general purpose processor
to implement a wireless engine core. One unique part of their design is the debug communication
link which supports a simple wireless protocol to the main data bus.

Given that modern day wireless transceivers consume several tens of mW [122], it is also
impractical to keep the transceiver active as this would greatly reduce the life expectancy of the
battery and the node. Therefore, a duty cycle scheme is often implemented that defines when the
node’s transceiver is active and when the node is in sleep mode, typically governed by timing
schemes in software. By realising that computing power can be more power efficient than
transmitting data, sensor nodes can carry out on-board processing to reduce the information
packet and transmission time. Data fusion and sensor network management fall hand-in-hand with
each other to achieve power efficiency [118]. Data fusion is where the removal of repeated data,
the use of min, max, or average values is used to reduce the number of transmissions.

Other hardware challenges involve reducing power consumption, reducing size, and reducing
costs. WSN designs can vary greatly for these goals and requirements can be met using FPGAs,
microelectromechanical systems (MEMs), and CMOS manufacturing technologies, as well as
systems engineering and integration. MEMs technologies is the use of nano-scale manufacturing
of microcentimetre size components such as processors, sensors, and actuators which are typically
integration on one piece of material. The Smart Dust project as described by B. W. Cook [119] is
a good example of CMOS and MEMS technologies integrated together in a 16 mm² autonomous
WSN. D. McIntire describes LEAP (Low Energy Aware Processing) in [120] which contains a
new architecture using the MPS430 and CC2420 devices together with an energy efficient
algorithm to reduce energy usage by sensing and characterising the environment. Their testbed
includes LEAP hardware, 802.11b board and imager. The imager here consumes 7 W (peak) and
the developed algorithm is only used sparingly to detect environment changes. Their future work
lies in software development, specifically dealing with the dynamic power expenditure during boot and how to handle multiple events.

### 2.5.1.2 Use of Wireless Sensor Networks in Space

A study, carried out for ESA and SSTL, studied wireless kits and investigated how they could be used for *intra-satellite* (on-board) and *inter-satellite* (between satellite) communication [108, 121, 122, 123, 124]. As described in Section 1, the use of COTS components undergoes rigorous environmental tests (including vibration, thermal and radiation) which were carried out on three mote kits: TelosB Motes, BTNodes, and Jennic High-Powered Modules, as shown in Figure 2-16.

![TelosB, BTNode, and Jennic High Power Module Motes undergoing sine vibration tests](image)

Various functional tests were also performed covering high data rate, low data rate, and long range scenarios to give an insight into various wireless sensor network hardware and software systems designs. These tests showed that each mote kit had its merits and faults where the Jennic motes had over 1 km range but at the cost of power and the TelosB motes had a very robust low power routing method as the cost of range. Thermal and vibration tests on these COTS mote kits showed no major concerns. But radiation tests showed that the memory devices used in these WSN motes produced *total ionic dose* (TID) damage, where material is ionised and charges are allowed to build up in large material areas such found in gates, allowing a reduced breakdown voltages. This is where the current draw is typically increased causing threshold shifts, leakage currents or even timing skews, especially bad for wireless communications.

From this study, many points to consider in the future can be taken:

1. **The radiation exposure produced total dose damage.** This is phenomenon is explained well in [125] where it is often the peripheral circuitry that fails (such as the charge pump circuit - used to add voltage to a particular line) and not the FLASH memory itself. Non-volatile memory should be targeted if radiation is an issue.
2. **WSNs display reliable long range communications in harsh environments.** Radio components with maximum performance at minimum mass could be targeted for use in space, such as that found in the Jennic mote tests, and in Surrey Space Centre's PCBSat [126].

3. **Footprint size costs power.** As the standard distributed computing platform is represented in layers, each layer will have varying power requirements and is especially important for distributed computing systems that use embedded devices, such as wireless sensor networks, where CPU and memory resources are limited. A good example of this involves tests using Jennic High Powered Modules that showed an increase in power consumption when the 802.15.4 network protocol is loaded into memory (from 55 mW to 132 mW).

4. **Functionality costs time.** All these extra software layers provide greater functionality to existing hardware at the cost of development time and power consumption. All these extra layers not only consume more power, but will also hinder functionality to non real-time applications like that found on the BTNode motes [122].

### 2.6 Summary

This chapter has investigated modern distributed computing techniques and there are some key findings that are relevant towards this research. Modern systems consider a layered model approach that often includes the TCP/IP software stack for communication. There are many paradigm models used such as the client/server, publish/subscribe, and peer-to-peer, and software Agents which can provide differing functionalities for different network scenarios and applications. Real-time systems are also discussed to conclude that hard real-time functionality is required for mission critical systems, such as those found in satellite/aerospace systems.

Software Agents thus far are not aimed at real-time systems as they operate autonomously, making them unpredictable, and are primarily enabled by Java based technologies, also nondeterministic due to large standard libraries, a slow and undeterminable execution model, and dynamic class loading execution times (to name a few reasons). Despite this Java offers a uniform platform that is taken forward with a review of real-time Java using just-in-time compilers and Java hardware processors which is highly desirable to develop new distributed computing middleware and applications. Agent implementations are reviewed to find that there are existing standards by FIPA and common frameworks, such as JADE, that can be used to handle complex distributed control or communication applications. But real-time applications of this technology are still maturing with no existing hard real-time solutions.

Experimentation of wireless sensor network hardware for intra and intersatellite communication also highlights that future embedded networked systems using COTS components need
consideration for radiation tolerance/mitigation. Another finding also suggests that more complex functionality will also cost both time and power which needs to be addressed when designing embedded networked systems with limited resources.
Chapter 3

3 Distributed Satellite Systems

This chapter explores the potential technologies that could be utilised for a distributed satellite system (DSS) demonstrator mission. Satellite missions and scenarios are discussed highlighting potential improvement areas related to this research. Section 3.1 introduces satellite constellations and discusses picosatellites and their current trends. Section 3.2 describes common DSS terms and reviews the current and future missions. Section 3.3 explains picosatellite orbit considerations and existing space perturbations/disturbance torques. Section 3.4 contains simulations of initial satellite deployment and 2 mission scenarios towards a stable constellation for DSSs. Section 3.5 highlights the use of modern distributed computing in space and discusses the motivation and applications for using Agents. Section 3.6 summarises the chapter’s findings.

3.1 Introduction

As described in Section 1.1, modern day satellite missions are costly, resource constrained in terms of mass and power, and undergo various environmental issues during their operation. But there is a trend towards using commercial-off-the-shelf components (COTS) and taking advantage of miniaturised components so that satellites can be built much smaller and at a fraction of the cost.

As an example, the Disaster Monitoring Constellation (DMC) [134] from takes advantage of multiple microsatellites in a constellation to achieve the impressive 24 hour revisit time to any location on Earth at a fraction of the cost. There are many other efforts at reducing spacecraft mass yet retaining functionality including:

- Nanosatellites such as PalmSat [135],
- Picosatellites such as the CubeSat platform [136], and
- Femtosatellites such as SpaceChip [137].

3.1.1 Picosatellites

As described in Chapter 1, a picosatellite is a class of satellite with a mass between 0.1 and 1 kg. Modern picosatellite design, development and launch has been made easier since the introduction
of the CubeSat standard by Stanford University and California Polytechnic Institute in 2000 [138]. From this point onwards, pico-satellites are only reviewed based on the CubeSat standard will be discussed as a representative class of pico-satellites.

Table 3-1 presents the results of a comprehensive review of all CubeSat missions (from 1U to 3U) containing information on the pico-satellite launch, current status, sub-systems implemented and mission drivers. This data is used to provide statistical analysis on the current pico-satellite design trends.

Various COTS parts such as structures, on-board computers (OBC), attitude systems [139], power systems [140], transceivers [141], and deployers with launch opportunity [142] are now available to the community of pico-satellite designers [143]. Advantages of using these COTS ‘kits’ includes a cheaper and faster development time (important for undergraduate and graduate students with limited time on their projects), some common interfaces and a community of developers. In April 2008, Pumpkin Inc sold their 100th unit of their Flight Module and Development Board supporting the growing trend in pico-satellite design [136].

The success of CubeSats launched so far vary in success rate as shown in Figure 3-1 which is based on data from Table 3-1. This is due to their fast design and build time with students but education systems are learning fast and success rates are increasing.

Figure 3-1. CubeSat Success Chart

A major impact on the success rate shown in Figure 3-1 is the Dnepr launch explosion in 2006 which destroyed 14 CubeSats or 43% of the total pico-satellite missions. Had this launch proved...
successful, a number of CubeSat designs and COTS components could be taken into consideration for future designs.
### Table 3-1. Picosatellite Comparison Review

<table>
<thead>
<tr>
<th>Satellite</th>
<th>Status</th>
<th>Specifications</th>
<th>Power (mW)</th>
<th>Mission</th>
</tr>
</thead>
<tbody>
<tr>
<td>CanX-l</td>
<td>Launch: 30/06/2003</td>
<td>OBC: ARM7 Core (Programmed in C/ No OS) + 512KB SRAM, 32MB Flash</td>
<td>400</td>
<td>1) Space Test of Sensors;</td>
</tr>
<tr>
<td>Uni. of Toronto, Canada</td>
<td>Failed: Unknown Prob</td>
<td>Com: Half-Duplex Transceiver (1.2 kbps)</td>
<td>TX 1100/ RX 150</td>
<td>CMOS horizon sensor + star tracker</td>
</tr>
<tr>
<td>Reference: <a href="http://www.utnicun.or.jp/cubesat/index-e.html">www.utnicun.or.jp/cubesat/index-e.html</a></td>
<td>ADCS: 3 x Honeywell Magnetorquer + Magnetometer</td>
<td>500-1000</td>
<td>GPS-based position determination</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Other: Mono + Color CMOS Imagers, COTS GPS Receiver</td>
<td>300</td>
<td>ARM7 CPU test</td>
<td></td>
</tr>
<tr>
<td>DUT Sat-I</td>
<td>Launch: 30/06/2003</td>
<td>OBC: Atmel AT91M40800 + eCos + 1MB RAM, 2MB Flash</td>
<td>155.1</td>
<td>1) MEMs Sun Sensor Test</td>
</tr>
<tr>
<td>Uni. of Demark, Denmark</td>
<td>Failed: Unknown Prob</td>
<td>Com: Half-Duplex Transceiver (2.4 kbps AFSK)</td>
<td>1350</td>
<td>2) 600m Tether Test</td>
</tr>
<tr>
<td>Reference: dutsat1.dutsat.dtu.dk</td>
<td>ADCS: Magnetorquers, 5 x sun sensors, magnetometer</td>
<td>160</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Other: Active pixel CCD cameras (Not implemented)</td>
<td>140</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AAU-CubeSat</td>
<td>Launch: 30/06/2003</td>
<td>OBC: Siemens C161 MCU + 4MB RAM</td>
<td>1084</td>
<td>1) Education</td>
</tr>
<tr>
<td>Aalborg University, Denmark</td>
<td>Partial Success: 1 month</td>
<td>Com: MX909 Modem + SC-450 Radio (9.6 kbps GMSK)</td>
<td>500</td>
<td>2) Communications</td>
</tr>
<tr>
<td>Reference: <a href="http://www.cubesat.auc.dk">www.cubesat.auc.dk</a></td>
<td>ADCS: 1 Magnetorquer, 1 sun sensor, 1 magnetometer</td>
<td>450</td>
<td>3) Colour CMOS image download</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Other: Active CMOS Imager (MCM20027 1.3 MP)</td>
<td>250</td>
<td>4) Test ADCS</td>
<td></td>
</tr>
<tr>
<td>Cute-I</td>
<td>Launch: 30/06/2003</td>
<td>OBC: H8/300 MPU (No OS)</td>
<td>Unknown</td>
<td>1) Communication</td>
</tr>
<tr>
<td>Tokyo Inst. Of Tech, Japan</td>
<td>Success: Still in operation</td>
<td>Com: DJ-C4 UHF downlink + DJ-C1 VHF uplink + CW beacon (400 mW)</td>
<td>TX 1100/ RX 100</td>
<td>2) Sensing</td>
</tr>
<tr>
<td>Reference: fss.mes.titech.ac.jp/ssp/cubesat/index_e.html</td>
<td>ADCS: 4 x Gyros, 4 x accelerometers, CMOS sun sensor</td>
<td>100</td>
<td>3) Deployment</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Other: CMOS Sun Sensor/ 2 x CCD Cameras</td>
<td>Unknown</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XI-IV</td>
<td>Launch: 30/06/2003</td>
<td>OBC: PIC 16F887 per sub-system + 32KB EEPROM</td>
<td>20</td>
<td>1) Command Up/Down Link</td>
</tr>
<tr>
<td>Uni. of Tokyo, Japan</td>
<td>Success: Still in operation</td>
<td>Com: Custom FM transmitter (1.2 kbps FSK AX.25) + CW Beacon</td>
<td>TX 6000/ RX &lt; 100 + 300</td>
<td>2) Telemetry Broadcasting</td>
</tr>
<tr>
<td>Reference: <a href="http://www.space.tu%C2%AD%C2%ADtokyo.ac.jp/cubesat/index-e.html">www.space.tu­­tokyo.ac.jp/cubesat/index-e.html</a></td>
<td>ADCS: No ACS (fixed magnet)</td>
<td>0</td>
<td>3) Imaging Experiment</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Other: Satellite sensors + CMOS image sensor</td>
<td>20 + 150</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XI-V</td>
<td>Launch: 27/10/2005</td>
<td>As above (backup for XI-IV)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uni. of Tokyo, Japan</td>
<td>Success: Still in operation</td>
<td>(Imager probe 2 months)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reference: <a href="http://www.space.t%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%C2%AD%E2%80%A6">www.space.t­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­…</a></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NuCube-2</td>
<td>Launch: 27/10/2005</td>
<td>OBC: Atmel AVR 8-bit RISC MCU</td>
<td>15</td>
<td>1) AIS - Automatic ID System</td>
</tr>
<tr>
<td>Uni. of Norway, Norway</td>
<td>Failed: Unknown Prob</td>
<td>Com: UHF Tx/Rx (9.6 kbps GMSK AX.25) + S-Band Transmitter</td>
<td>1000</td>
<td>2) Receive, Store, Retransmit to Ship</td>
</tr>
<tr>
<td>Reference: <a href="http://www.nucube.no">www.nucube.no</a></td>
<td>ADCS: 3 x Magnetorquers + 1 x HMR2300 magnetometer + boom</td>
<td>228</td>
<td>3) Reindeer Herd Monitoring</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Other: AIS</td>
<td>Unknown</td>
<td>(Similar to Ncube-1)</td>
<td></td>
</tr>
<tr>
<td>UWE-1</td>
<td>Launch: 27/10/2005</td>
<td>OBC: H82764R MPU (uCLinux) + 8MB RAM + 4MB Flash</td>
<td>300</td>
<td>1) Technology Development:</td>
</tr>
<tr>
<td>Würzburg, Germany</td>
<td>Partial Success: 3 weeks</td>
<td>Com: Telemetry, Tracking and Command UHF (9.6 kbps AFSK)</td>
<td>1000</td>
<td>uCLinux Tests</td>
</tr>
<tr>
<td>Reference: www7.informatik.uni-wuerzburg.de/cubesat</td>
<td>ADCS: Unknown</td>
<td>Unknown</td>
<td>New Ga As Solar Cells</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Other: Unknown</td>
<td>Unknown</td>
<td>2) Integration of Internet protocols</td>
<td></td>
</tr>
</tbody>
</table>
Table 3-1. Picosatellite Comparison Review (Continued)

<table>
<thead>
<tr>
<th>Satellite</th>
<th>Status</th>
<th>Specifications</th>
<th>Power (mW)</th>
<th>Mission</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CUTEL7 + APD</strong></td>
<td>Launch: 22/02/2006</td>
<td>OBC: ARMV41 (Windows CE.NET) + 32MB RAM + 128MB Flash</td>
<td>Unknown</td>
<td>1) Use Complicated PDA and protect from radiation</td>
</tr>
<tr>
<td>Tokyo Inst. Of Tech, Japan</td>
<td>Success: Still in operation (1.2kg + 0.8kg APD)</td>
<td>Com: VHUF Uplink + UHF Downlink (1.2/ 9.6 kbps GMSK AFSK) + Beacon</td>
<td>Tx 300/ Rx ? + 100</td>
<td>2) Test Avalanche Photodiode charged particle detector</td>
</tr>
<tr>
<td>Reference: lss.mes.titech.ac.jp/ssp/cute1_7/index_e.html</td>
<td></td>
<td>ADCS: 1 x Gyro sensor + 1 magnetometer + 1 magnet sensor</td>
<td>&gt; 46</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Other: CMOS Camera Module + S-Band Digimeter</td>
<td>Unknown</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>ION</strong></td>
<td>500</td>
<td>1) Measuring Oxygen emissions</td>
</tr>
<tr>
<td>Uni. of Illinois, USA</td>
<td>Failed on Dnep Launch</td>
<td>OBC: Hitachi SH2 7045 (Custom Linux based OS) + 1MB SDRAM + 8MB F</td>
<td></td>
<td>2) Microvacuum arc thruster (µVAT) 4W</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ADCS: Honeywell magnetometer + torquer + 4 x micro vacuum arc thruster</td>
<td>120 (Trusters unknown)</td>
<td>4) CMOS Camera (640x480/ 0.3 mpix)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Other: Photomultiplier tube (PMT) + CMOS camera (0.3 MP)</td>
<td>150</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Sacred</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uni. of Arizona, USA</td>
<td>Failed on Dnep Launch</td>
<td>OBC: MCU</td>
<td>Unknown</td>
<td>1) TID Radiation Experiment</td>
</tr>
<tr>
<td>Reference: cubesat.arizona.edu/sacred_sat</td>
<td></td>
<td>Com: (1.2 kbps AFSK AX.25)</td>
<td>Tx 3000/ Rx ?</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ACDS: Unknown</td>
<td>Unknown</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Other: Unknown</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(Poor Website)</td>
<td><strong>KUTEsat</strong></td>
<td>47</td>
<td>1) Education + Inter University Relations</td>
</tr>
<tr>
<td>Kansas, USA</td>
<td>Failed on Dnep Launch</td>
<td>OBC: PIC16F819 MCU (ucDimm Dragonball)</td>
<td>Tx 1800/ Rx S400</td>
<td>2) Miniature Manouevring Control System</td>
</tr>
<tr>
<td>Reference: <a href="http://www.engr.ku.edu/ae/kutesat.htm">www.engr.ku.edu/ae/kutesat.htm</a></td>
<td></td>
<td>Com: Receiver/ Transceiver</td>
<td>500</td>
<td>3) Prototype/ test JPL Solar Sail or NASA</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ACDS: Honeywell HMC2003 + magnetometer + sun sensor</td>
<td>4 + 1500</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Other: CMOS Camera + Dosimeter</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>ICEcube1</strong></td>
<td>180</td>
<td>1) GPS scintillation science through ionosphere</td>
</tr>
<tr>
<td>Cornell, NY, USA</td>
<td>Failed on Dnep Launch</td>
<td>OBC: Motorola 68HC05 MCU + 1MB RAM + 256KB EEPROM</td>
<td>Tx 2782/ Rx 224</td>
<td>(Identical to ICEcube2)</td>
</tr>
<tr>
<td>Reference: <a href="http://www.mae.cornell.edu/cubesat">www.mae.cornell.edu/cubesat</a></td>
<td></td>
<td>Com: Custom MCU Transceiver (9.6 kbps FSK AX.25)</td>
<td>234 to 1500</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ACDS: 3 x Magnetorquers + AeroFin</td>
<td>2000 @ 25% duty</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Other: Cougar GPS</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>ICEcube2</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cornell, NY, USA</td>
<td>Failed on Dnep Launch</td>
<td>As above (identical to ICEcube1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reference: <a href="http://www.mae.cornell.edu/cubesat">www.mae.cornell.edu/cubesat</a></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>RINCON 1</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uni. of Arizona, USA</td>
<td>Failed on Dnep Launch</td>
<td>OBC: Unknown</td>
<td>Unknown</td>
<td>1) Establishing Communications</td>
</tr>
<tr>
<td>Reference: bache.arizona.edu/ axtorwiki/index.php/Rincon_1</td>
<td></td>
<td>Com: Unknown</td>
<td>Tx 400/ Rx ? + 10</td>
<td>2) Sensing Mission</td>
</tr>
<tr>
<td></td>
<td>(Poor Website)</td>
<td>ACDS: UHF Up/Down (1.2 kbps AFSK AX.25) + Similar Beacon</td>
<td>Unknown</td>
<td>3) Orbit. ADCS Analysis Mission</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Other: Unknown</td>
<td>Unknown</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>SEEDS</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nihon University, Japan</td>
<td>Failed on Dnep Launch</td>
<td>OBC: MPU per sub-system</td>
<td>Unknown</td>
<td>1) Establishing Communications</td>
</tr>
<tr>
<td>Reference: cubesat.acro.ost.nihon-u.ac.jp/english/main_e.html</td>
<td></td>
<td>Com: FM/ CW Transmitter</td>
<td>Tx 450/ 90</td>
<td>2) Sensing Mission</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ACDS: Gyro + magnetic field sensors</td>
<td>Unknown</td>
<td>3) Orbit. ADCS Analysis Mission</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Other: Digital talker + large sensor subsystem (temp + current)</td>
<td>Unknown</td>
<td></td>
</tr>
<tr>
<td>Satellite</td>
<td>Status</td>
<td>Specifications</td>
<td>Power (mW)</td>
<td>Mission</td>
</tr>
<tr>
<td>---------------------</td>
<td>-------------------------</td>
<td>-------------------------------------------------------------------------------</td>
<td>------------</td>
<td>------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Ncube-1</td>
<td>Failed on Dnepr Launch</td>
<td>OBC: Atmel AVR 8-bit RISC MCU</td>
<td>15</td>
<td>1) AIS - Automatic ID System</td>
</tr>
<tr>
<td>Uni of Norway, Norway</td>
<td></td>
<td>Com: UHF TxFxRx (9.6 kbps GMSK AX.25) + S-Band Transmitter</td>
<td>1000</td>
<td>2) Receive, Store, Retransmit to Ship</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ADCS: 3 x Magnetorquers + 1 x HMR2300 magnetometer + boom</td>
<td>228</td>
<td>3) Reindeer Herd Monitoring</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Other: AIS</td>
<td>Unknown</td>
<td>(Similar to Ncube-2)</td>
</tr>
<tr>
<td>MEROPE</td>
<td>Failed on Dnepr Launch</td>
<td>OBC: Motorola MC68HC812A4 (H12) MCU + 150KB RAM</td>
<td>429</td>
<td>1) Radiation tests in the Van Allen belts</td>
</tr>
<tr>
<td>Montana State Univ, USA</td>
<td></td>
<td>Com: Yaesu VX-1R Transceiver VHF Down (9.6 kbps) + UHF Up (1.2 kbps)</td>
<td>Tx 3000/ Rx 1286</td>
<td>(Note: Geiger Tube req &gt; 500V, current stated at 5mA = 2.5W)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ADCS: 2 x Fixed Magnets</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Other: Geiger Tube (+ Collimator)</td>
<td>480</td>
<td></td>
</tr>
<tr>
<td>AeroCube-1</td>
<td>Failed on Dnepr Launch</td>
<td>OBC: Unknown</td>
<td>Unknown</td>
<td>1) Test communication &amp; bus system</td>
</tr>
<tr>
<td>Company: Aero Corp., USA</td>
<td></td>
<td>Com: Omnidirectional patch antenna</td>
<td>2.000</td>
<td>2) Test camera suite</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Other: Suite of CMOS cameras</td>
<td>Unknown</td>
<td></td>
</tr>
<tr>
<td>CP1</td>
<td>Failed on Dnepr Launch</td>
<td>OBC: BX24 MCU (Basic) + 400B RAM + 32KB EEPROM</td>
<td>100</td>
<td>1) Test Sun Sensor</td>
</tr>
<tr>
<td>California Poly, USA</td>
<td></td>
<td>Com: Alinco DJ-C5 Transceiver (1.2 kbps FSK AX.25) + UHF CW Beacon</td>
<td>300 to 500</td>
<td>(Optical Energy Tech)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ADCS: Magnetorquers + magnetometers + sun sensor</td>
<td>Unknown</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Other:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CP2</td>
<td>Failed on Dnepr Launch</td>
<td>OBC: PIC18 MCU + 256KB Flash</td>
<td>Unknown</td>
<td>1) Energy Dissipation Experiment</td>
</tr>
<tr>
<td>California Poly, USA</td>
<td></td>
<td>Com: 2 x PIC18 + 2 x CC1100 + 2 x RFMD2117 (1.2 kbps AFSK AX.25)</td>
<td>1000</td>
<td>2) Test Standardised Bus for CPX</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ADCS: 6 x HMC1052 Magnetometers + 6 x magnetorquers</td>
<td>990</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Other: Dummy Payload (testing configuration)</td>
<td>300</td>
<td></td>
</tr>
<tr>
<td>Mea Huaaka</td>
<td>Failed on Dnepr Launch</td>
<td>OBC: RabbitCore 2000 + PIC per sub-system</td>
<td>300</td>
<td>1) Active antenna</td>
</tr>
<tr>
<td>Uni. of Hawaii, Hawaii</td>
<td></td>
<td>Com: Yaesu FT-847 Transceiver</td>
<td>1.500</td>
<td>2) Thermal sensors</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ADCS: Permanent magnet + hysteresis rod</td>
<td>0</td>
<td>3) Attitude stabilization</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Other: COTS CMOS Imager</td>
<td>Unknown</td>
<td></td>
</tr>
<tr>
<td>GeneSat-1</td>
<td></td>
<td>OBC: PIC (Custom Code) + FLASH</td>
<td>Unknown</td>
<td>1) Perform experiment on E. Coli bacteria</td>
</tr>
<tr>
<td>Space Tech Lab, CA, USA</td>
<td></td>
<td>Com: Beacon: MHX-2400 (172 kbps) + Beacon (1.2 kbps AFSK AX.25)</td>
<td>Tx 4380/ Rx 1150</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Success (Mis. Complete)</td>
<td>ADCS: Permanent magnets + hysteresis rods</td>
<td>Unknown</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Other: Microfluid experiment + heater</td>
<td>1250</td>
<td></td>
</tr>
<tr>
<td>AeroCube-2</td>
<td>Partiial Success</td>
<td>OBC: Unknown</td>
<td>Unknown</td>
<td>1) Test Comms System</td>
</tr>
<tr>
<td>Company: Aero Corp., USA</td>
<td></td>
<td>Com: 902-928 MHz (9600 bps GFSK)</td>
<td>2000</td>
<td>2) Systems Bus</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ADCS: None</td>
<td>0</td>
<td>3) Multiple CMOS Cameras</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Other:</td>
<td>Unknown</td>
<td></td>
</tr>
</tbody>
</table>
Table 3-1. Picosatellite Comparison Review (Continued)

<table>
<thead>
<tr>
<th>Satellite</th>
<th>Status</th>
<th>Specifications</th>
<th>Power (mW)</th>
<th>Mission</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAPE-1</td>
<td>Launch 17/04/2007</td>
<td>OBC: PIC18F (C code) per subsystem</td>
<td>Unknown</td>
<td>1) TTC Mission</td>
</tr>
<tr>
<td>Uni. Louisiana, USA</td>
<td>Success: Still in operation (Telemetry beacon non operational, CW beacon semi-operational)</td>
<td>Com: CC1020 ICs UHF (9.6 kbps FSK AX.25)</td>
<td>1000</td>
<td>2) Education</td>
</tr>
<tr>
<td>Reference: cape.louisiana.edu</td>
<td>ACDS: Permanent Magenets</td>
<td>Other: Camera</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Unknown</td>
<td></td>
</tr>
<tr>
<td>California Poly, USA</td>
<td>Success: Still in operation</td>
<td>Com: CC1000 UHF Transceiver (1.2 kbps FM FSK AX.25)</td>
<td>1000</td>
<td></td>
</tr>
<tr>
<td>Reference: polysat.calpoly.edu/CP1.php</td>
<td>ACDS: Magnetometers + magnetorquers</td>
<td>Other: None</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>CP4</td>
<td>Launch 17/04/2007</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>California Poly, USA</td>
<td>Partial Success</td>
<td>OBC: PICI8F8720</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reference: polysat.calpoly.edu/CP2.php</td>
<td>(OBC failure after 2 months – possible IC problem)</td>
<td>Com: CC1000 UHF Transceiver (1.2 kbps FM FSK AX.25)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ACDS: Magnetometers + magnetorquers</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Other: None</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CSTB1</td>
<td>Launch 17/04/2007</td>
<td>OBC: 4 x MCUs</td>
<td>Unknown</td>
<td>1) Testbed for future Boeing Satellites</td>
</tr>
<tr>
<td>Company: Boeing, USA</td>
<td>Success: Still in operation</td>
<td>Com: UHF Transceiver (1.2 kbps FM AFSK AX.25)</td>
<td>1000</td>
<td>2) Redundant Radios</td>
</tr>
<tr>
<td></td>
<td>Mass: 1.5kg in 1U</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LIBERTAD-1</td>
<td>Launch 17/04/2007</td>
<td>OBC: MSP430 (Pumpkin Board)</td>
<td>66</td>
<td>1) Camera</td>
</tr>
<tr>
<td>Reference: <a href="http://www.usrgerioarboleda.edu.co/proyecto_cesapcial">www.usrgerioarboleda.edu.co/proyecto_cesapcial</a></td>
<td>ACDS: Unknown</td>
<td>Other: Batteries Li Ion</td>
<td>Unknown</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(Website in Spanish)</td>
<td></td>
<td>Unknown</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MAST</td>
<td>Launch 17/04/2007</td>
<td>OBC: PIC18F8720 + 128KB memory</td>
<td>100</td>
<td>1) Multi-Application Survivable Tether</td>
</tr>
<tr>
<td>Company: Tethers Unlimited</td>
<td>Success: Still in operation</td>
<td>Com: ISM Band (2.4GHz FHSS)</td>
<td>1000</td>
<td>2) Takes pics as tether is deployed and the CubeSats separate</td>
</tr>
<tr>
<td>Reference: <a href="http://www.tethers.com/Missions.html">www.tethers.com/Missions.html</a></td>
<td>ACDS: 1km Holytether TM</td>
<td>Other: GPS on Gadget (STTL SGR-05)</td>
<td>Unknown</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(1U + 1U + 1U called Ted, Gadget, Ralph)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AAUsat-2</td>
<td>Launch 28/04/2008</td>
<td>OBC: ARM7 MPU (Custom C) + 2MB RAM + 8MB Flash (code/data)</td>
<td>&lt; 300</td>
<td>1) Comms to satellite</td>
</tr>
<tr>
<td>Aalborg, Demark</td>
<td>Success: Still in operation</td>
<td>Com: Custom UHF Transceiver (1.2/ 9.6 kbps AFSK FSK AX.25)</td>
<td>Tx 2000/ Rx 283</td>
<td>2) ACDS + Gamma Ray Payloads</td>
</tr>
<tr>
<td>Reference: aauasi.space.aau.dk/homepage/index.php?language=en&amp;page=home</td>
<td>ACDS: Magnetorquers + momentum wheels + sun sensors + gyro + more</td>
<td>Other: Imager (1.3 MP) + Motorola MCM20027</td>
<td>Unknown</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(Recovered from SEE 11th March 2009)</td>
<td></td>
<td>250</td>
<td></td>
</tr>
<tr>
<td>Satellite</td>
<td>Status</td>
<td>Specifications</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-------------------</td>
<td>-------------------------</td>
<td>--------------------------------------------------------------------------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CanX-2</td>
<td>Launch 28/04/2008</td>
<td>OBC: ARM7 MPU + 2MB SRAM + 16 MB Flash + TMR</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Success: Still in operation</td>
<td>Com: UHF Transceiver + S-Band Transmitter (32/256 kbps GMSK AX.25)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ACDS: Sun sensors + wheels + propulsion</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Other:</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3U CubeSat (3.5 kg)</td>
<td>Unknown</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Mission: 1) Test new equipment: propulsion system momentum wheels, GPS receiver, sun sensor</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>2) Atmospheric spectrometer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Compass One</td>
<td>Launch 28/04/2008</td>
<td>OBC: 4 (PIC, 8051, DPS) + 16 MB Flash</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Success: Still in operation</td>
<td>Com: 437,405 MHz (1.2 kbps AF SK AX.25 + 2.4/4.8 kbps MSK)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ACDS: Magnetometer, sunsensors + magnetorquers</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Other: CMOS Imagier</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Mass: 3 kg</td>
<td>Unknown</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Mission: 1) GPS Receiver (800 mW)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>2) High speed ground link</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>3) CMOS Imagier</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CUTE 1.7 + APD II</td>
<td>Launch 28/04/2008</td>
<td>OBC: ARMV41 (Windows CE.NET) + 32MB RAM + 128MB Flash</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tokyo Inst. Of Tech, Japan</td>
<td>Success: Still in operation</td>
<td>Com: VHF Uplink + UHF Downlink (1.2/9.6 kbps GMSK AF SK) + Beacon</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reference: <a href="http://www.cubesat.de/">www.cubesat.de/</a></td>
<td></td>
<td>ADCS: 1 x Gyro sensor + 1 magnetometer + 1 Sun &amp; Earth sensor</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Other: CMOS Camera Module + S-Band Digimeter</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Mass: 3 kg</td>
<td>66</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Mission: 1) New methods using PDA + bus</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>2) Education + collaboration</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Delfi-C3</td>
<td>Launch 28/04/2008</td>
<td>OBC: MSP430 (Pumpkin Board)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Delph University, Denmark</td>
<td>Success: Still in operation</td>
<td>Com: Custom VHF Up + UHF Down (1.2 kbps BPSK AX.25)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reference: <a href="http://www.delfic3.nl/index">www.delfic3.nl/index</a> php?option=com_content&amp;task=view&amp;id=27&amp;Itemid=113</td>
<td>ACDS: Sun sensor + magnetic hysteresis rods</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Other: No batteries</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Mass: 3 kg (Transponder failure confirmed 29th Jan 2009)</td>
<td>Unknown</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1) Autonomous sun sensing</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>2) Thin film solar cells</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>3) on-board wireless (915 MHz)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SEEDS (2)</td>
<td>Launch 28/04/2008</td>
<td>OBC: 4 x MCU</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nihon University, Japan</td>
<td>Success: Still in operation</td>
<td>Com: Custom VHF Up + UHF Down (1.2 kbps AFSK AX.25) + CW Beacon</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reference: cubesat.cpsi.niho n-u.ac.jp/english/seeds_2_e.html</td>
<td>ACDS: gyromagnetic sensors + gyro sensors thermal</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Other: UHF Digitalinker</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Mass: 3 kg</td>
<td>Unknown</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1) CW Communications</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>2) FM Packet Download</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>3) Digitaltalker Operation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CP6</td>
<td>Launch 15/05/2009</td>
<td>OBC: PIC18LF6720 + 128B RAM + 128KB external memory</td>
<td></td>
<td></td>
</tr>
<tr>
<td>California Poly, USA</td>
<td>Success: Still in operation</td>
<td>Com: CC1000 UHF Transceiver (1.2 kbps FM FSK AX.25)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reference: polysat.calpoly.edu/CP6.php</td>
<td>ACDS: Magnetometers + magnetorquers</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Other: None</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Mass: 450 /Rx ? + 110</td>
<td>Unknown</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1) Three Axis ACDS Mission Testing</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>2) Deployment of spring steel tapes</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>3) Navy Electron Collection device</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HawkSat-1</td>
<td>Launch 15/05/2009</td>
<td>OBC: MSP430 (Pumpkin Board)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hawk Institute for Space Sci., USA</td>
<td>Success: Still in operation</td>
<td>Com: Unknown</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reference: <a href="http://www.hawkspace.org/index.html">www.hawkspace.org/index.html</a></td>
<td>ACDS: Unknown</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Mass: 1000</td>
<td>Unknown</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1) Customer testing new components exposed to radiation + temperature of space</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>2) Deployment of spring steel tapes</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>3) Navy Electron Collection device</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AeroCube 3</td>
<td>Launch 15/05/2009</td>
<td>OBC: Unknown</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Company: Aero Corp., USA</td>
<td>Success: Still in operation</td>
<td>Com: CC1101 900-928 MHz ISM Band (38.4 kbps)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reference: <a href="http://www.aero.org">www.aero.org</a></td>
<td>ACDS: Unknown</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Mass: 66</td>
<td>Unknown</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1) Technology Demonstration</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>2) Deployment of spring steel tapes</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>3) Navy Electron Collection device</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3-1. Picosatellite Comparison Review (Continued)
Table 3-1. Picosatellite Comparison Review (Continued)

<table>
<thead>
<tr>
<th>Satellite</th>
<th>Status</th>
<th>Specifications</th>
<th>Power (mW)</th>
<th>Mission</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pharmasat</td>
<td>Launch 15/05/2009</td>
<td>OBC: PIC (Custom C code) + FLASH</td>
<td>Unknown</td>
<td>1) Perform experiment on E. Coli bacteria</td>
</tr>
<tr>
<td>NASA, USA</td>
<td>Success: Mis. Complete</td>
<td>Com: Beacon: MHX-2400 (172 kbps) + Beacon (1.2 kbps AFSK AX.25)</td>
<td>Tx 4380/Rx 1150</td>
<td></td>
</tr>
<tr>
<td>Reference:</td>
<td>Mass: &lt; 5 kg, 3U CubeSat</td>
<td>ACDS: Permanent magnets + hysteresis rods</td>
<td>Unknown</td>
<td></td>
</tr>
<tr>
<td>tia.arc.nasa.gov/pharmasat</td>
<td>Other: Microfluid experiment + heater</td>
<td>1250</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note: Information marked as ‘unknown’ is where data could not be found. This is due to a poorly maintained or non-existent website, classified or commercial information, or language/translation problems.
3.1.2 Picosatellite Design Trends

In this section, CubeSat designs are compared since 2003 and the main findings from Table 3-1 are summarised here. All subsystems and their power consumption were investigated including the onboard computer, communications, attitude determination and control, and payload systems.

3.1.2.1 On Board Computer

There are many types of onboard computer that have been implemented in differing missions and include utilising microprocessors (MPUs) such as the ARM7 and microcontrollers (MCUs) such as the PIC or Siemens C161.

There are two major differences between MPUs and MCUs: instruction set and inputs/output (I/O) resources. MPUs typically have a large instruction set for performing many functions making them flexible in functionality whilst MCUs have a dedicated small instruction set for set peripherals or I/O interfaces. MCUs however have incorporated interfaces and memory into 1 device whilst MPUs still need other devices to support the memory and interfacing that often increases power consumption.

Figure 3-2 shows the evolution of OBC choices made on all picosatellite missions so far based from data in Table 3-1.

![Figure 3-2. Evolution of OBC Systems (Successes & Failures)](image)

Figure 3-2 shows that the most common OBC choice lies with the microcontroller as they have heritage in space where 32-bit MCU systems can handle many types of missions and at varying
low power modes. Even though MPUs can offer more generic functionality, the power consumption is one of the driving reasons why they are not chosen for CubeSat designs.

With reference to Table 3-1, there are some developers which have attempted terrestrial computing systems in space including DTU Sat-1 uses Atmel with eCos operating system that failed [144], UWE-1 used H8 Series with uCLinux which had some success but was noted to have very 'buggy' software [145], and CUTE1.7 + APD uses ARMV4I with WinCE.NET which uses two modified personal data assistants (PDAs) [146].

KuteSat [147] also operated uCDimm: a commercial version of uCLinux. It was chosen for hardware interfaces, scheduler, shell, IP networking, real-time clock but the kernel and file system alone was 850 kB. A large proportion of communications and command scheduler software for telemetry is broadcast over AX.25, a commonly used data link layer protocol for amateur radio systems, using an IP address. Data was encrypted and authenticated between the satellite and ground where users can execute programs remotely. Known software issues on KUTESat included large applications, large libraries, bugged sub-system PIC programs, and problems fitting data on the chosen 4 MB flash. This project was an important step in realising configurable and terrestrial software on a picosatellite because if the communications and other systems could be implemented in software, it could be uploaded and reconfigured after launch.

Also BeeSat, to be launched on the Vega launcher in late 2009, also aims to use a middleware called BOSS to be run on a Philips LPC2292 microcontroller [148]. This aims to provide transparent software/hardware interaction without needing any additional device drivers through a subscribe protocol.

### 3.1.2.2 Communications

All of the CubeSat missions operate in the same way with a transmitter and receiver. Some satellites such as the XI-IV and XI-V [149] have an additional broadcast transmitter for amateur radio broadcasts to Earth. They all have power consumptions between 350 mW to 1 W depending on what is being sent (e.g. image data, satellite housekeeping data, etc). It is often found that communication only occurs when orbiting over certain areas for mission control to down/ upload information.

More recent developments have seen the use of IEEE 802.15.4 (commonly called ZigBee) radios launched and confirmed working on CAPE1 [150] typically used in wireless sensor networks. This is extremely significant because it proves that COTS wireless communication devices can be used to communicate in space at high speeds.
3.1.2.3 Electrical Power System

Solar power and rechargeable batteries are predominantly the main power source for all the CubeSat satellites and a number of systems solutions are available. Variables in number of batteries, environmental protection/assembly (temp, etc), lifespan, charge/discharge capacity are all possible areas for further study. Solar panels vary from COTS components to custom designs for differing efficiencies from between 16 – 25% efficiencies.

3.1.2.4 Attitude and Determination Control System

There have been 3 identifiable systems implemented in CubeSat missions taken from data found in Appendix 3:

- Full ADCS – Custom/COTS components used for manoeuvrability of the satellite (increases power consumption from 160 mW to 1 W).
- Partial ADCS – Sensor only (reduced power consumption to < 25 mW).
- Passive ADCS – Use of a boom or permanent magnet (more mass and addition of a mechanical system for the boom).

This system is fully determined by the power budget and payload requirements of the picosatellite.

3.1.2.5 Payload Options

CubeSats payloads are typically chosen based on their missions which often driven by educational or teaching purposes because they are cost effective to build and customise for many novel mission scenarios as shown in Figure 3-3 obtained from the data in Table 3-1. This cost effectiveness also reduces the financial risk involved by educational facilities and companies.

![Figure 3-3. Picosatellite Mission Comparison](image)
With any space mission, each design surveyed has multiple objectives and Figure 3-3 shows that the majority of CubeSat missions are either primarily technological or communications motivated, specifically testing new space sensors or new architectures to getting a simple link from space to ground. Common payloads include CMOS Cameras or Active Pixel CCD Cameras as well as solar weather sensors such as the sun sensors used on Cute-l mission [151].

### 3.2 Distributed Satellite Missions

A *Distributed Satellite System* (or DSS) is defined as a system of multiple satellites designed to work together in a coordinated fashion to perform a mission [152]. A DSS consists of two or more satellites that are distributed in space and form a co-operative infrastructure for measurement data acquisition, processing analysis and distribution. DSSs do not need to link directly to other companion satellites and can be free to make independent observations. An example scenario concept is found in Figure 3-4 and can be expanded to include tens or even hundreds of satellites, as envisioned by NASA’s ANTs Mission [153].

As shown in Figure 3-4, by using multiple satellites together, redundancy is introduced into the mission. For example, if one from the group is damaged, the mission could still be completed with the remaining satellites. Other advantages include increasing temporal and spatial data by having more sensors at a particular location and time but also enabling differing satellites to achieve differing tasks armed with different instruments such as imagers, gamma ray sensors, etc.

Some common terms are briefly overviewed below:

- An *Intersatellite Link* (ISL) is the use of communication between satellites for a specific functionality (e.g. routing data to the ground or relative ranging).
A Cluster is a functional grouping of spacecraft, formations, or virtual satellites. A Formation is a multiple spacecraft system with desired position and/or orientation relative to each other or to a common target.

Formation Flying is the term used for the tracking and maintenance of a desired relative separation, orientation or position between or among spacecraft. Although no formation-flying missions have been demonstrated in orbit at present, research on formation flying is very active focusing primarily on the control, communication and navigation issues.

A Virtual Satellite is a spatially distributed group of satellites working as a single unit to perform a specific mission using intersatellite links, one example being an array configuration to provide a large aperture.

A Space Sensor Networks is the use of multiple very small satellites for multi-point local sensing using intersatellite links to 'sink' payload or telemetry data to ground.

A historical review of DSSs is given by D. Barnhart in [3].

3.2.1 Current Distributed Satellite Missions

This section reviews the current systems already designed or in orbit that aid towards enabling distributed operations using distributed computing and intersatellite links.

The IRIDIUM satellite constellation [154, 155] is a wireless personal communications network designed to permit a wide range of telephone services-voice, data, fax, paging to connect to destinations virtually anywhere on earth. Sixty six operational satellites at 662 kg are configured in six near-polar orbital planes, in which 11 satellites circle in one plane to track the location of the telephone handset (this allows IRIDIUM to cover all areas of the Earth). Each satellite operates 7 PowerPC CPUs to control routing and voice compression drawing 620W. This system is the only example of a DSS where on-board computing and intersatellite links are utilised to complete a mission. Despite being a technological success, the project was an economic failure [156].

Emerald [157, 158, 159] is a two satellite mission to explore “Robust Distributed Space Systems” between the Space Systems Development Lab (SSDL) at Stanford University and Santa Clara Remote Extreme EnvironMent (SCREEM) lab at Santa Clara University. Here, a distributed bus architecture without ISLs is used for on-board distributed computing. The “smart” subsystems/peripherals contain their own computational power on an I²C bus and data bus, allowing the main MCU to take on a co-ordinating role, free from the burden of menial tasks. Comparable to object-oriented computer programming, this ‘object-oriented hardware’ uses these
Chapter 3. Distributed Satellite Systems

subsists to perform data processing at the source (e.g. at sensors/payloads). Emerald was never launched.

The Cluster mission is $4 \times 1200$ kg spacecraft to gather detailed data for a three-dimensional map of the magnetosphere [160]. The satellites are each in different orbits and collaboratively collect data when all satellites pass the same area without intersatellite link capabilities. This configuration has been providing scientific data and breakthroughs. All computing is done on ground, offloading any online computations from the satellites.

TechSat-21 [161] [162]: The Autonomous Sciencecraft Constellation flight demonstration (ASC) was to fly onboard the Air Force’s TechSat-21 constellation of $3 \times 150$ kg satellites. The onboard flight software includes several interesting software components, notably the use of ObjectAgent and TeamAgent cluster management software that enables Techsat-21 spacecraft to autonomously perform manoeuvres and high precision formation flying to form a single virtual instrument. This mission was cancelled in 2001 due to the problem complexity and project overruns.

Milstar [163] is the most advanced military communications satellite system to date that provides secure, jam resistant, worldwide communications to meet essential wartime requirements for high priority military users (such as ships, submarines, aircraft and ground stations). The $4,500$ kg satellites operate intersatellite link capability. The operational Milstar satellite constellation consists of six satellites positioned around the Earth in geosynchronous orbits with one non-operational. Launched over 9 years from 1994 to 2003, each satellite serves as a smart switchboard in space by directing traffic from terminal to terminal anywhere on the Earth, like a LAN network. Since the satellite actually processes the communications signal and can link with other satellites through intersatellite links, the requirement for ground controlled switching is significantly reduced. It has the ability to interface many communications protocols via point-to-point and point-to-multi-point communications.

NASA’s ST5 [164] was launched on 22nd March 2006 to act as buoys for monitoring the weather of space and the enormous storms spawned by the sun. The three 55 lb ST5 microsatellites were hauled into a highly elliptical orbit aboard an air-launched Pegasus rocket (inclined 105.6 degrees). Designers have packed 10 advanced technologies into the satellite, including an X-Band transponder communication system which is smaller than current systems for command, telemetry, and tracking communications between the microsatellites and ground stations.

The transponder is low-power (5.5 W @ 7.2 V DC) and low-weight (862 g). It also uses less power than previous flight-proven systems using a high power amplifier, diplexer, band pass filter, and two X-band antennas [165]. This mission aims to give the X-band transponder some space qualification and testing for future missions. It is not used for intersatellite links.
The FORMOSAT-3/COSMIC [166] constellation of six (70 kg) microsatellites was launched successfully from Vandenberg Air Force Base in California on Friday, 14th April 2006. COSMIC’s (also “Constellation Observing System for Meteorology, Ionosphere and Climate”) mission is to collect atmospheric sounding data for scientific research and operational testing for constellation design and analysis, development of the spacecraft bus and payload instrument development. There are no ISLs on this mission either.

All these discussed missions are summarised in Table 3-2.

Table 3-2. Comparison of Current Distributed Satellite Systems

<table>
<thead>
<tr>
<th>Mission</th>
<th>Distributed Satellite System</th>
</tr>
</thead>
<tbody>
<tr>
<td>IRIDIUM</td>
<td>Uses ISLs, distributed telecoms routing, stable constellation, high cost</td>
</tr>
<tr>
<td>Emerald</td>
<td>No ISLs, on-board distributed computing, never launched</td>
</tr>
<tr>
<td>Cluster</td>
<td>No ISLs, no on-board distributed computing, varied constellation periods</td>
</tr>
<tr>
<td>TechSat-21</td>
<td>Uses ISLs, distributed formation control, never launched, high cost</td>
</tr>
<tr>
<td>Milstar</td>
<td>Uses ISLs, distributed routing, stable constellation, high cost (military)</td>
</tr>
<tr>
<td>NASA ST-5</td>
<td>No ISLs, stable constellation, no on-board distributed computing</td>
</tr>
<tr>
<td>FORMOSAT-3/</td>
<td>No ISLs, stable constellation, no on-board distributed computing</td>
</tr>
<tr>
<td>COSMIC</td>
<td></td>
</tr>
</tbody>
</table>

Table 3-2 summarises the discussed DSSs focusing on key features for a distributed satellite system: the use of ISLs, the use of on-board distributed computing, the constellation design, and the cost. Each constellation has some progress towards a complete distributed satellite system but no current mission meets all the requirements towards intersatellite communications with multiple satellites, on-board processing, and the utilisation of smaller cost effective platforms.

3.2.2 Future Distributed Satellite Mission Challenges

The current DSS missions have been huge missions with high costs in terms of deployment and in orbit maintenance. The current commercial and scientific missions typically do not attempt:

- Intersatellite connectivity due to the high fuel/propellant cost to keep a formation or constellation.
- On-board processing for data aggregation and collection or on-board autonomy for constellation management.

Three missions extend the current DSS mission capability: NASA’s ANTS Framework [167,168], DARPA’s F6 Program [169], and Surrey Space Centre’s (SSC) PCBSat concept [126]
The NASA ANTS Framework is a concept mission based on Addressable Reconfigurable Technology (ART) adaptable for the full spectrum of space activities. ART systems based on currently available electromechanical (EMS) technology could support human crews on the lunar surface within the next 10 to 15 years. The Prospecting Asteroid Mission (PAM) application, the first application considered for ANTS architecture requires MEMS level application of the ANTS architecture. This technology is not yet available but is estimated for 20 years time. It is keeping with long-term Exploration Initiative plan for robotic exploration of the next target, the main-belt asteroids, beyond Mars.

On the 26th February 2008, the Defence Advanced Research Projects Agency (DARPA) also awarded over $38.5 million to 4 companies in the US to demonstrate the fractionation of a large monolithic spacecraft through the use of smaller and wireless satellite cluster modules that act in a distributed and co-ordinated way. This “future, fast, flexible, fractionated, free flying spacecraft” project (also known as the F6 program) aims to develop fractionated modules which can provide unique functionality to the cluster. Key technologies have been identified requiring development shown in Figure 3-5.

![Figure 3-5. DARPA F6 Program Key Technologies](image)

Various meetings have already started the brainstorming of such technologies, including a presentation from R. Golding and T. Wong on using Agents as a means for communication and control [170].

To realise a space sensor network, a ‘satellite-on-a-chip’ and ‘satellite-on-a-PCB’ (or PCBSat) concepts were investigated at Surrey Space Centre by D. Barnhart [126]. PCBSat was designed and built at Surrey Space Centre towards detecting plasma bubbles in the ionosphere, see Figure 3-6.
The computational capability of this femtosatellite shown in Figure 2-6 is limited to microcontroller based computations. Some additional networking functionality is also enabled using the ZigBee radio systems to sink data to an orbiting ‘master’ satellite (such as a picosatellite) using ad-hoc wireless links.

One topic so far un-discussed is the intersatellite link properties such as range, data-rate, and power consumption. K. Sidibeh [171] targets an RF range between 500 to 2000 km for wireless connectivity between microsatellites using IEEE 802.11 at 1W with smart antennas. Here, the IEEE 802.11 standard is adapted for use in LEO constellations towards a wide local area network (WLAN) in example string-of-pearl constellation scenarios.

Future concepts include a joint EADS Astrium and ESA project target a range of 25 m to 250 m for the microsatellite SMART-2 in 2011 and 2015 [172], without a discussion on orbital parameters or mission length. Both of these solutions aim at using small satellites with a design and launch costs between $5-50 Million, which makes the total constellation costs significantly higher than using a smaller satellite bus.

### 3.3 Picosatellite Constellations and Orbital Considerations

As picosatellite designs are notably cheaper through the use of COTS sub-systems and faster to build to the small mass, picosatellite constellations are investigated for developing future distributed satellite system constellations at a fraction of the cost, previously described in Section 3.1.1 and 3.1.2. When looking at any constellation aiming to use intersatellite links, important orbital factors to consider are:

1. **Initial Conditions:** The classical orbital elements \((a, e, i, \Omega, \omega, \mu)\)
2. **Satellite Mass & Volume
3. **Relative Range and Speeds between satellites
4. Access times from satellite to satellite(s)

The classical orbital elements describe a satellite's motion around a central body (typically the Earth) which can be used to provide intersatellite ranging/access indicators for collaborative opportunities, shown in Figure 3-7.

![Classical Orbital Elements](image)

**Figure 3-7. Classical Orbital Elements**

### 3.3.1 Orbital Perturbations and Disturbance Torques

Orbital perturbations are forces that change the satellite's classical orbital elements and disturb the satellites from the nominal orbit. There are three types of variation as described in [173] and [174]: secular drift, short-period variations (variations in the elements less than or equal to the orbit period), long-period variations (variations greater than the orbital period). Secular variations have long term effects on the orbit parameters caused by the gravitational forces of the Sun and Moon, non-spherical mass distribution of the Earth (or oblateness) and atmospheric drag.

The oblateness of the Earth, also known as the J2 effect (including J3, J5 and J7 as the main terms), changes the right ascension of the ascending node, \( \Omega \), and moves the argument of perigee, \( \omega \) (a shift in the direction that gravity pulls on the satellite). The movement of the right ascension of the ascending node, \( \Delta \Omega \), moves the satellite west for prograde orbits (\( i < 90^\circ \)), east for a retrograde orbits (\( i > 90^\circ \)) and zero for polar orbits (\( i = 90^\circ \)). The movement of the argument of perigee, called the perigee rotation rate, \( \dot{\omega} \), rotates the perigee locations (in the same direction of the satellites motion if \( i < 63.4^\circ \) but \( > 116.6^\circ \) or the opposite direction if \( i \) between 63.4° and
116.6°). Both these effects can collate together to large perturbations but can be used practically in our advantage in sun-synchronous orbits or Molniya orbits.

Atmospheric drag takes orbital energy from the satellite by friction with Earth’s upper atmosphere and will be the biggest problem facing sensor nodes in space. This orbital energy reduction is a function of the semi-major axis, a, so this is also decreased. As the satellite passes through the atmosphere, the apogee of the orbit will also change with a negative ΔV leading to a more circular orbit (reducing the eccentricity). Eventually, as the orbit gets lower with each orbit, the satellite will decay and re-enter the Earth’s atmosphere.

Solar radiation pressure is the force exerted on an object from the Sun’s emission of electromagnetic radiation in the form of photons (or energy particles). When photons emitted from the Sun hit a satellite, they exert a small force that could disturb a satellite’s orientation and orbit over a long period of time. This pressure is very small when compared to atmospheric drag, measured at 5N per 1 km². For picosatellite designs, this force is often negligible and ignored as the mission life expectancy is short and the surface area is very small [175]. An overview of the discussed orbital perturbations are shown in Figures 3-8.
3.4 Satellite Constellations and Simulations

As described in Section 3.2, a distributed satellite system requiring intersatellite links could be formed for a number of application scenarios. But for each application, specific orbits would be required to meet the mission goals whilst still taking advantages of intersatellite links. In this section, two case-study scenarios are investigated: a string-of-pearl constellation and a flower constellation. A cluster or formation scenario is also discussed. These missions are summarised in Table 3-3 which highlights some of the characteristics for each constellation designs and their potential application domains, dependent on each mission’s intersatellite communication needs.

<table>
<thead>
<tr>
<th>Constellation</th>
<th>Characteristics</th>
<th>Applications</th>
</tr>
</thead>
<tbody>
<tr>
<td>String-of-Pearl</td>
<td>• Polar/ sun-synchronous orbits</td>
<td>• Earth or Space Observation</td>
</tr>
<tr>
<td></td>
<td>• Predictable connection periods</td>
<td>• Communication</td>
</tr>
<tr>
<td></td>
<td>• Limited mobility</td>
<td>• Global Positioning/Navigation</td>
</tr>
<tr>
<td>Flower</td>
<td>• Elliptical orbits</td>
<td>• Science</td>
</tr>
<tr>
<td></td>
<td>• Predictable connection periods</td>
<td>• Multi-point Atmospheric/Space Weather Monitoring</td>
</tr>
<tr>
<td></td>
<td>• Known mobility patterns</td>
<td>• Distress Beacon Monitoring</td>
</tr>
<tr>
<td>Cluster</td>
<td>• Similar orbits</td>
<td>• Experimental Orbits for Earth Observation, Communication and Positioning</td>
</tr>
<tr>
<td></td>
<td>• Unpredictable connection periods</td>
<td>• Hardware Fractionation</td>
</tr>
<tr>
<td></td>
<td>• Medium/ high mobility. Unknown patterns</td>
<td>• Multi-point Atmospheric/Space Weather Monitoring</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Earth Observation, Communication and Positioning</td>
</tr>
</tbody>
</table>

The next section investigates issues in initial deployment from the rocket and distributed mission operation opportunity in 3 scenarios: the string-of-pearls, the Flower constellation, and a cluster mission, investigating their performance towards distributed missions.

3.4.1 Initial Ejection from Launcher

Figure 3-9 depicts the cumulative realisation of putting a distributed computing system in a standard sun synchronous low Earth orbit.
Chapter 3. Distributed Satellite Systems

1. When initially deployed, the satellites will be close together where communication network links are good.

2. As the satellites progress in their orbits, they will be affected by torques/disturbances and communication links will be stretched as their orbital energies change for each piconode.

3. After many orbits, the satellites will be affected by many perturbations and will drift apart. Once the satellites are out of range of the server, the network is lost.

Figure 3-9. Perturbation Affects on Distributed Mission

Depicted in Figure 3-9, the satellites orientation changes through solar radiation pressure, atmospheric drag and other disturbances will cause the satellite to lose connection with the master satellite or cluster head in ad-hoc network terms. Other perturbations include the Earth’s magnetic dipole, aerodynamics, and gravity gradients but are considered negligible or not applicable for the scenarios presented here.

These particular issues also relate to the exit velocities in modern day picosatellite deployers where each satellite exits the deployer at differing velocities and with differing vector angles to each other. Any velocity vector difference means that the satellites separate quicker. To find the relative ranges between the satellites when deployed classical orbital elements are converted to the Cartesian co-ordinate reference frame (x, y, z) using Satellite Toolkit (STK) [176] and making changes to each satellite’s velocity vector using the following equations where $V_{\text{mag}}$ is the vector magnitude, and X, Y, and Z are the velocities in the X, Y, and Z axis correspondingly:

$$V_{\text{mag}} = \sqrt{X^2 + Y^2 + Z^2}$$

(3.1)

Where the unit vector $\hat{X}$, $\hat{Y}$, and $\hat{Z}$ are the respective vector magnitudes, given by:

$$\hat{X} = \frac{X}{V_{\text{mag}}} , \, \hat{Y} = \frac{Y}{V_{\text{mag}}} \, \text{and} \, \hat{Z} = \frac{Z}{V_{\text{mag}}}$$

(3.2)

By adding vector magnitude, $V_{\text{mag}}$, and the desired change to the vector magnitude, $V_{\text{change}}$, then multiplying by the previous respective vector magnitudes, the new vector magnitudes can be given as $\hat{X'}$, $\hat{Y'}$, and $\hat{Z'}$:

$$\hat{X'} = (V_{\text{mag}} + V_{\text{change}}) \times \hat{X} , \hat{Y'} = (V_{\text{mag}} + V_{\text{change}}) \times \hat{Y} \, \text{and} \, \hat{Z'} = (V_{\text{mag}} + V_{\text{change}}) \times \hat{Z}$$

(3.3)
Chapter 3. Distributed Satellite Systems

Figures 3-10 and 3-11 conclude that the relative ranges of the 3 satellite mission of picosatellites at separation rates of 1 cm/s and 5 cm/s respectively in a sun-synchronous orbit showing that they quickly separate over a period of weeks due to perturbations.

Typically, each picosatellite is ejected from the deployer with the intention of drifting apart very quickly to avoid collision and achieve ground communications. Whereas, here, close attention is paid to the relative ranges to keep satellites together for a distributed scenario. An investigation into the deployer springs under space environment conditions would be needed to find the exact mechanical characteristics to realise these separation rates.
3.4.2 In-Orbit Constellation Scenarios

This section presents the simulation results of two constellation orbit scenarios: the string-of-pearls constellation and the Flower constellation, investigating their performance towards distributed missions.

Orbit simulation parameters used include a mass of 1 kg, a volume of 10 cm³, and a ballistic coefficient of $5.92 \times 10^{-16}$, found using the equation below:

$$B = \frac{1}{2} C_D \frac{a}{m} \rho$$  \hspace{1cm} (3.4)

Using the following values for the parameters in Equation 3.4, $C_D = 2.2$ (flat plate model), cross sectional area, $a = 20$ cm² (for a tumbling CubeSat), a mass, $m = 1$ kg, and atmospheric density, $\rho = 2.961 \times 10^{-13}$ kg/m³ (Harris-Priester model for a maximum atmospheric density at 680 km [177]), the ballistic coefficient is found below:

$$B = \frac{1}{2} \times 2.2 \times \frac{0.002}{1} \times 2.691 \times 10^{-13} = 5.92 \times 10^{-16}$$

More detailed satellite properties including the inertia matrix is given if Table 3-4 for the picosatellite during simulation.

<table>
<thead>
<tr>
<th>Satellite Property</th>
<th>Min</th>
<th>Expected</th>
<th>Max or Tolerance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Launch Mass (kg)</td>
<td>5.25</td>
<td>5.25</td>
<td>100</td>
</tr>
<tr>
<td>Centre of Gravity</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Xg (mm)</td>
<td>216.63</td>
<td>216.63</td>
<td>+/- 10</td>
</tr>
<tr>
<td>Yg (mm)</td>
<td>0</td>
<td>0</td>
<td>+/- 10</td>
</tr>
<tr>
<td>Zg (mm)</td>
<td>6.560</td>
<td>6.560</td>
<td>250</td>
</tr>
<tr>
<td>Moment of Inertia</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ixx (kg/m²)</td>
<td>0.01689</td>
<td>0</td>
<td>+/- 1.0</td>
</tr>
<tr>
<td>Iyy (kg/m²)</td>
<td>0.3317</td>
<td>0</td>
<td>+/- 1.0</td>
</tr>
<tr>
<td>Izz (kg/m²)</td>
<td>0.3295</td>
<td>0</td>
<td>+/- 1.0</td>
</tr>
</tbody>
</table>

The values in Table 3-4 are used in the STK simulation to give accurate orbital simulation [178].

3.4.2.1 String-of-Pearls Constellation Scenario

The *string-of-pearls* constellation is where several picosatellites are launched in the same orbit, spaced along the orbit, so that they come over a target sequentially for greater ground coverage.
The investigation of the string-of-pearls constellation using STK looks at groups of picosatellites in similarly inclined orbits to produce various formations, reproduced in by the classical orbital elements found in Table 3-5 and shown in Figure 3-12.

### Table 3-5. String-of-Pearls Constellation Classical Orbital Elements

<table>
<thead>
<tr>
<th>Satellite</th>
<th>RAAN (deg)</th>
<th>Mean &amp; True Anomaly (deg)</th>
</tr>
</thead>
<tbody>
<tr>
<td>111</td>
<td>9.38</td>
<td>90</td>
</tr>
<tr>
<td>112</td>
<td>6.97</td>
<td>83.64</td>
</tr>
<tr>
<td>113</td>
<td>0</td>
<td>81.02</td>
</tr>
<tr>
<td>114</td>
<td>353.03</td>
<td>83.64</td>
</tr>
<tr>
<td>115</td>
<td>350.62</td>
<td>90</td>
</tr>
<tr>
<td>116</td>
<td>353.03</td>
<td>96.36</td>
</tr>
<tr>
<td>117</td>
<td>0</td>
<td>98.98</td>
</tr>
<tr>
<td>118</td>
<td>6.97</td>
<td>96.36</td>
</tr>
<tr>
<td>119</td>
<td>0</td>
<td>90</td>
</tr>
</tbody>
</table>

Table 3-5 gives the exact orbit parameters required to reproduce the simulation shown Figure 3-12 visualises the constellation and the multiple orbital planes. The relative ranges in the string-of-pearls constellation are propagated using STK’s High Precision Orbital Propagator (HPOP) and is shown in Figure 3-13. It can be clearly seen that for longer periods of time, this constellation does not stay close to each other for performing a distributed satellite mission due to perturbation effects previously discussed in Section 3.3.1. Satellites in slightly differing orbital planes can
however drift secularly together as shown by the relationship between picosatellite 115 and picosatellite 119 which oscillates between 200 to 1000 km (shown in red).

Figure 3-13. Year long simulation on example String-of-Pearls

Although this drift pattern aids in keeping a tighter formation, there is a very real risk of collision also, as shown in Figure 3-142 where satellites 113 and 114 get dangerously close to the central picosatellite 119 at 0 km (shown in green and yellow respectively).

Figure 3-14. 3 Month Simulation on example String-of-Pearls

To conclude, the string-of-pearls constellation is predictable and provides short-term range of each other, but the perturbations which cause the picosatellites to drift apart gives a low mission lifetime for performing distributed satellite mission that would require expensive orbit maintenance to achieve.
3.4.2.2 Flower Constellation Scenario

The *Flower constellation* is regarded as being a stable formation to perform satellite missions within orbit. Applications proposed and initially investigated include GPS missions, reconnaissance, two-way orbits, multiple science missions and planetary exploration, explained in greater detail in [179].

Upon closer investigation, there are some distinct features including [180]:

- The constellation’s axis of symmetry coincides with the spin axis of the Earth.
- Each satellite has the same orbit shape (anomalistic period, argument of perigee, height of perigee and inclination).
- Satellites are equally displaced along the equatorial plane to complete the constellation using the right ascension of the ascending node (RAAN), true anomaly or mean anomaly.

A revised methodology of setting up a Flower constellation for LEO is outlined in Appendix C. The following paragraphs extend the current knowledge at validating the Flower constellation by including perturbations (up to J44), atmospheric drag, solar radiation pressure, and tumbling effects through detailed STK HPOP simulations. The Flower constellation is previously only described as mathematical patterns, including only up to J2 perturbations. The orbital parameters and picosatellite phasing for 9 picosatellites are listed in Table 3-6 and shown visually in Figure 3-15 with drawn ISLs from picosatellite F-1 to all other picosatellites. All the nodes are separated equally for all picosatellites using the mean anomaly and true anomaly.

### Table 3-6. Flower Constellation Orbital Parameters and Phasing for a 9 Picosatellite Flower Constellation

<table>
<thead>
<tr>
<th>Sat #</th>
<th>Mean Anomaly (deg)</th>
<th>True Anomaly (deg)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0.00</td>
</tr>
<tr>
<td>2</td>
<td>40</td>
<td>53.54</td>
</tr>
<tr>
<td>3</td>
<td>80</td>
<td>98.12</td>
</tr>
<tr>
<td>4</td>
<td>120</td>
<td>134.10</td>
</tr>
<tr>
<td>5</td>
<td>160</td>
<td>165.20</td>
</tr>
<tr>
<td>6</td>
<td>200</td>
<td>194.80</td>
</tr>
<tr>
<td>7</td>
<td>240</td>
<td>225.90</td>
</tr>
<tr>
<td>8</td>
<td>280</td>
<td>261.88</td>
</tr>
<tr>
<td>9</td>
<td>320</td>
<td>306.46</td>
</tr>
</tbody>
</table>
Chapter 3. Distributed Satellite Systems

Figure 3-15. LEO Flower Constellation

The constellation results shown in Figure 3-16 and Figure 3-17 exhibit sets of equidistant oscillating picosatellite pairs, shown at 4 month and 4 days measurements respectively.
This Flower constellation implementation gives a constant and predictable range from any given satellite. Unlike polar orbit constellation simulations, the Flower constellation ensures picosatellites will drift secularly together along the Earth’s equator, keeping them in formation for a much longer, estimated to be from 3-6 months to over 10 years for greater and more cost-effective operations or science return.

The access time between each picosatellite is proposed as the best metric to predict distributed collaboration. The access time is the time for 2 satellites to communicate between each other dependent on a set range. Figure 3-18 shows the access time between for a range of a requirement, 400 km, with a picosatellite constellation (1 kg or less) showing satellites drift in and out of range at different times.
Chapter 3. Distributed Satellite Systems

The communication range of 400 km is arbitrarily chosen as the lowest range from the literature in Section 3.2.2 and assumed to give sufficient collaborative opportunity. Access times between picosatellites range between 3 days to 14 days dependent on the main sink picosatellite in the constellation which could be used for controlling distributed operations. The sink picosatellite is the satellite chosen to communicate to ground because if all satellites tried to communicate to ground, the link would be overrun (assuming 1 operational frequency). If the sink satellite is no longer the optimal central satellite or if the server satellite is damaged/cannot perform its role, a new satellite or configuration is needed. But while these findings give predictable and repeating connection patterns, it is important to note that the exact times for satellite collaboration can never be guaranteed when including the aforementioned perturbations and disturbances.

Formation flying and clustering missions discussed here can be useful where the mission requires images or data readings from differing local locations, also noted in Table 3-3. Atmospheric and space weather sensing missions could benefit from several data points. The technologies developed towards these missions could be used in future planetary space exploration missions. But formation flying at this stage is a niche segment of LEO based missions, which is often not required. A satellite’s attitude (or orientation) for intersatellite communication could also be important but is not explored in this research. An omni-directional radio/communications system is assumed but a combined pointing and ADCS may be vital in the future.

3.5 Distributed Computing in Space

The idea to use terrestrial distributed computing methods in space is not new and use of Internet Protocol (or IP) has been demonstrated with the client/server model on a number of missions, summarised in Table 3-7.

<table>
<thead>
<tr>
<th>Mission Name</th>
<th>Date</th>
<th>Technology Demonstrated</th>
</tr>
</thead>
<tbody>
<tr>
<td>STRV-1b</td>
<td>1996</td>
<td>First Satellite with an IP Address (in 120 kB RAM) [181]</td>
</tr>
<tr>
<td>UoSAT-12</td>
<td>2000</td>
<td>First IP Stack demonstrating pinging, clock synchronisation, FTP and UDP packets [182]</td>
</tr>
<tr>
<td>Cabletron Router on ISS</td>
<td>Feb. 2001</td>
<td>First VoIP [183]</td>
</tr>
<tr>
<td>CHIPSat</td>
<td>Jan. 2003</td>
<td>First to use TCP/IP for end-to-end communication [184]</td>
</tr>
<tr>
<td>AISat-1</td>
<td>Nov. 2002</td>
<td>CCSDS File Delivery Protocol [185]</td>
</tr>
<tr>
<td>UK-DMC</td>
<td>Sept. 2003</td>
<td>CLEO mobile access router alongside imaging payloads</td>
</tr>
</tbody>
</table>
Further use of IP is also found on Beijing-1 (Oct. 2005), MidSTAR-1 (Mar. 2007) and CFESat (Mar. 2007). In all the missions shown in Table 3-7, the groundstation is the server and a single satellite is the client. For multiple satellites to be used, all would require an existing middleware (or virtual environment) with supporting hardware and software to perform more complex missions such as formation flying or spacecraft fractionation.

The Consultative Committee for Space Data Systems (CCSDS) [186] has also been involved in several areas towards distributed computing in space, more notably their orange books in next generation space Internet (NGSI) architectures such as supporting spacecraft IP mobility using Agents from April 2003 [187]. Another perspective of distributed computing in space can be taken from the Delay Tolerant Network Research Group (DTNRG) which studies the communication strategies required in delay and disruptive ad-hoc networks [188]. This working group also contributed heavily to the Bundling protocol to identify end-point IP addresses in a given network for space networking and Saratoga, a protocol based on UDP to provide reliable transfers using checksums [79]. The code for these projects is currently implemented in C++ with RTEMS OS with a footprint of 512 kB and also in Java for simulation.

The state of the art methods for satellite communication has utilised TCP/IP, used commonly for the Internet, enabling over 20 years of terrestrial research and applying it for space. But the protocol still requires modification and the standard high performance through wire is not found in long distance wireless. M. E. Elaasar [189] sums up the unique challenges of using COTS wireless transport protocols in space and how to handle bit corruption, handoffs and limited connectivity. K. Sidibeh’s [190] proposes modifying the 802.11 standard for increased connectivity between satellites in LEO using smart antennas shows that timing changes in the MAC layer will enable high performance connectivity (minimal interference and maximising gains), even in changing satellite topologies.

### 3.5.1 Applied Agents for Space Applications

NASA’s Deep Space One (or DS1) was the first satellite that used an Intelligent Agent to control a spacecraft on an experiment called Remote Agent eXperiment (or RAX) in 1999 [194]. RAX was used to remotely plan, execute and reconfigure various schedules and modes of the spacecraft over a period of 2 days from the 19th May 1999 autonomously. TechSat-21 however was the first proposed satellite system encountered to suggest using Agent technology in space using multiple spacecraft [195].

T. Scheetter [196] suggests that clusters of satellites flying in formation require some level of onboard autonomy to fly within specified tolerance levels for collision avoidance, address fault detection, knowledge sharing and planning/scheduling. The paper goes on to explain
ObjectAgent, developed over the course of several U.S. Air Force small business innovative research (SBIR) programs by Princeton Satellite Systems. While ObjectAgent is a general-purpose application, the TeamAgent software is an extension of ObjectAgent, specifically applied to space-based formation flying. Foreseeable problems include their mission of a sparse-array aperture to synthesize a large radar antenna. This concept has a disadvantage whereby skilful positioning of the elements of the TechSat cluster means that position and timing uncertainties inherent in separately orbiting objects can limit performance.

The Messaging Architecture for Networked and Threaded Applications (MANTA) middleware [197, 198], previously called ObjectAgent in this previous work by T. Scheeter, looks into the relative motion problems in a distributed satellite system identified for relative navigation, initialisation, coarse/precise reconfiguration, pre-emptive/reactionary collision avoidance and fault monitoring programs. Although not completely discussed, a software architecture is proposed and 3 levels of autonomy are described for reconfigurable, decentralised guidance and control architecture for formation flying missions. The MANTA project has since been dropped.

The identification of spacecraft level Agents is described by J. B. Mueller in [199] towards levels of capable ‘intelligence’, also shown in Figure 3-19. The paper shows preliminary results on reconfiguring cluster formations of lead/follower satellite orbits using control systems using Agents.

Figure 3-19. Intelligence and Co-ordination Levels for Increasing Agent Use

Remote procedure call and Agent systems can be tested fully on this platform and an investigation into how Agent duplication on a spacecraft would affect the software resources where limited network connectivity would instantiate multiple agents on one system, also known as ‘software replication’.
H. Tripp discusses some of the problems in distributed operations in a cluster and suggests using Agents with stigmergy for co-ordination and management tasks [200], as found in Figure 3-18. For performing task allocation in a cluster, he also highlights that the client/server paradigm is not scalable and high latency as well as the fact that peer-to-peer requires complex negotiation over intersatellite links. He proposes that stigmergy is used to adapt individual node behaviours without intersatellite links based on layers of management and collaboration to the ground as shown in Figure 3-20.

The problem with this configuration is that when a formation cluster of local satellites contacts a groundstation, only 1 satellite can communicate down. Each satellite would have to wait for the channel to be free (or operate on differing frequencies) before sending to ground and therefore intersatellite links would be required to route data to a sink node, and then to ground.

3.5.2 Motivation towards the Agent Paradigm

Examples of spontaneous collapse of distributed networks [201] include well documented telecommunications outages where router software upgrades (having passed scaled-down test-bed examinations) malfunctioned in real life causing large scale outages. Router timing errors only emerged during fully operational interactions, something test-bed examinations did not pick up. This is also true in a distributed computing environment on multiple mobile nodes where complex algorithms are being implemented to perform a multiple mission goals and overcome connectivity issues associated with LEO satellites. A. Carzaniga’s paper [202] presents some interesting future uses of Agent systems using mobile code for distributed applications. Some applications elaborated here include:
Deployment, Upgrading and Maintenance: The exploitation of mobile code to support software deployment and maintenance is very useful in a highly distributed environment. Mobile code paradigms would allow the action of installation of new software units (such as IP blocks in an FPGA) or rebuilding of an application with new parameters locally at each satellite without the need of communications to individual picosatellites from ground. Proactive reconfiguration by comparing both hardware characteristics and software features for configuring and installation could be automated. Functionality could be added or removed dependent on the satellite’s resources. E.g. after critical testing and simulation, a fault is found in the communication software which limits the downlink’s data rate. Therefore a software patch can be uploaded to the current server satellite for distribution.

Customisation of Services: Agents on board satellites could provide a flexible service for both server and client roles dependent on their current hardware and software status. This allows Agents to be reactive in their communications to other satellites. E.g. If there are limited resources for the collection of sensor data on a client satellite, it will require storage on the network. It can communicate with the closest satellite, which identifies it as a server satellite and cannot be used for storage but reserve power for ground link. The server satellite can then suggest the next closest. Communicating with this satellite identifies itself as another client satellite with a damaged payload and can provide storage services to the rest of the network. Data transfer starts and data collection can continue.

Disconnected Operations: Identified physical links include a novel smart antenna system (low power with distances up to 10 km using 802.11 IEEE standards) [203] and a standard AX.25 communications system (typical with CubeSats) for downlinks, see Appendix A. Without an attitude determination and control system, the picosatellites could be tumbling and moving in differing orbits making intersatellite communications difficult. Unreliable links with low-bandwidth and low-reliability require new methods for allowing flexibility in the system. Improving granularity in the client-server paradigm would allow a greater number of operations but would increase local resource use, complexity and reduce flexibility for change.

Agents by their autonomous nature are designed to handle complex and unreliable networking. Research in how to manage power for communications and memory could make efficient use of the resources on-board. E.g. a satellite is disconnected and saves the data to transmit securely in a ‘reduced state’ in static memory allocation instead of large dynamic memory allocation.

Improved Fault Tolerance: Traditional client-server systems request data and services from remote environments which can then be executed locally. But if a return result is only partially complete, there are problems in addressing the state and possible recovery. As Agents encapsulate all state information in a single component, the information can be traced and recovered.
Destructive space environment effects could be mitigated using Agents to detect hardware or even software errors.

3.6 Summary

This chapter primarily looks at the areas of picosatellites, distributed satellite systems, and what technologies have been used towards distributed computing in space.

With the advent of the CubeSat standard and availability of various commercial off the shelf sub-systems, picosatellite designs are very popular – especially for educational establishments. They can be built and launched at a fraction of the cost of large missions. A major review of all picosatellite missions finds that current on-board computing use microcontrollers for low cost/power. The majority of missions currently launched are focused at technology demonstration of components and sensors for gaining flight heritage.

Distributed satellite systems terms are overviewed to find that there many satellite scenarios under review: namely clustering, formation flying, virtual satellites (also called fractionation), and space sensor networks. These new satellite systems are analogous to a highly mobile wireless sensor network as they have very minimal computing resources to sense and communicate a given environment. When overviewing the current distributed missions, only one mission offers intersatellite connectivity with on-board processing (Iridium) in a stable constellation. All future distributed missions discussed are all technology development exercises in intersatellite connectivity and on-board processing (and optimisation thereof).

The deployment of picosatellites is simulated to find that this is a key area in employing a distributed mission and two constellation designs were developed: string-of-pearl and the Flower constellation. Without orbit maintenance, the string-of-pearl design found perturbation issues are too great but the Flower constellation offers a stable scenario where satellites oscillate between each other giving a periodically intermittent link.

A popular distributed computing technology used in space includes the terrestrial TCP/IP stack and custom delay tolerant network protocols which uses end point IP addresses. Further review of Agent systems for space concludes that the full benefits of autonomy, code mobility, and distributed operation control are not utilised. Only simple scheduling (RAX) and suggested on-board autonomy software (TechSat-21) are investigated.
Chapter 4

4 Agent Computing Platform

This chapter looks at combining the key findings from the terrestrial domain in Chapter 2 and the space domain in Chapter 3 to derive the requirements for a new computing platform for distributed satellite systems. Section 4.1 highlights the previous chapter and summarises the key drivers towards an Agent computing platform in distributed space mission applications. Section 4.2 proposes a new Agent-based distributed computing platform utilising a Java co-processor and embedded Agent middleware.

4.1 Drivers and Requirements

This section highlights the previous two chapter summaries and what future drivers are needed towards distributed satellite systems. These areas and their technological drivers are Agents, space, and distributed space systems:

- To date, terrestrial distributed computing systems use Java for TCP/IP based networking applications (Section 2.3) and modern Agent technologies use a Java environment which provides services for control/communication applications (Section 2.4.4). Embedded solutions for networked distributed computing software are highly motivated to utilise less memory for reducing overall power consumption (Section 2.5).

- Space systems are sized based on a specific payload or mission application. It is then optimised for low mass to reduce the cost of launch (Chapter 1). They typically operate on low power to meet stringent power requirements and need to ensure reliability against the space environment (Chapter 1, Section 2.5.1.1).

- Distributed space systems have many technological drivers such intersatellite links/networking to computing resource sharing for performing formation flying or clustering missions for greater science return/€ (Chapter 1). Mobility support from existing efforts can drive new techniques and technologies for greater fault resilient/tolerant operations (Section 3.4 and 3.5).

From these drivers, requirements are derived for a new distributed computing system, broken into node level (individual satellite level) and network level (multiple satellite level) requirements:
**Node Level** Functionality Requirements:

1. Power optimised distributed computing platform using:
   
   1.1. **Low Footprint** (in ROM or Flash Memory).
   
   1.2. **Low Active Memory** usage (in RAM).
   
   1.3. **Ensure normal satellite operations** can occur with minimal overhead.

2. Storing and forwarding of data using distributed computing paradigms, including:

   2.1. **High Data** or **High Priority** Applications using a client/server paradigm such as payload data through the network such as imaging data, larger science payloads & data aggregation.

   2.2. **Low Data** Applications using a peer-to-peer (P2P) paradigm such as location and velocity changes such as telemetry, “byte” size payload data (GPS, science payload measurements) & network management data.

   2.3. **Enable Future Applications** for distributed operations, autonomy and artificial intelligence based on current terrestrial software systems whilst allowing legacy code.

**Network Level** Functionality Requirements:

3. Ad-hoc intersatellite networking capabilities for initial formation.

4. Adaptable and redundant ground-link communication schemes, i.e. main ‘sink’ to ground.

5. Proactive and reactive topology schemes to mobility.

These requirements should demonstrate the developed technologies to show a NASA technology readiness level (TRL) of 4 or above, proof of concept, critical analysis, and validation under lab conditions towards environmental testing on a testbed platform.

### 4.2 Design Approach

To meet the distributed computing requirements, discussed in Section 4.1, for a number of distributed satellite system scenarios, a SoC design is proposed for ‘node’ and an additional Java co-processor for ‘network’ operations as shown in Figure 4-1. Open source SoC processor solutions are used with the aim to run state-of-the-art distributed computing applications using Agents.
Figure 4-1 describes the basic proposed system where 1 processor core operates ‘node’ level functions whilst another processor core operates ‘network’ functionality. As shown in Section 2.5.1.2 and 2.5.2, it is often the case that additional software layers are added to existing software layers which costs development and process time. Using this method, each core and its software is decoupled from the other providing improved development speed than a standard processor system. This configuration however creates a dependency between each processor and if one fails, the whole system could fail thus reducing overall fault tolerance.

In relation to the drivers and requirements in Section 4.1, information towards solving each requirement is expanded further below:

1. **Hardware Architecture**: A hardware implementation of a general purpose processor for existing node functionality and the Java co-processor, JOP, is selected to enabling existing Java-based Agent middleware and networking.
   1.1. JOP is constrained to a Java runtime environment revision 1.1.8 and any software developed must conform to this requirement.
   1.2. Hardware logic overheads should be minimal and all software footprint sizes must be comparable to existing systems in ROM/Flash and RAM usage.
   1.3. To test the proposed Agent computing platform, a picosatellite testbed platform is designed based on the CubeSat standard. This will provide a set of resource limitations that the design must conform to and includes available power and mass values given potential orbits.

2. **Middleware Layer**: To complement client/server and peer-to-peer paradigms, an Agent based middleware is proposed allowing both future and legacy code to be embedded in
Agents and distributed or migrated as services. Existing terrestrial technologies have been used in space for ground link communications. A preferable scheme used is UDP, as described in [204] as it is better suited for 'store-and-forward' communications. A dropped UDP packet, in this case, is preferred to a TCP delayed packet but not in all situations. Therefore, the requirements are disseminated even further:

2.1. 'High Data Rate’ or ‘High Priority Data’ tasks to use TCP/IP for reliable and secure point-to-point communication.

2.2. ‘Low Data Rate’ uses UDP for relatively fast, broadcast/multicasting of small of amounts of data to groups of satellites as a publish/subscribe paradigm.

Both types can take advantage of existing Agent Communication Languages (ACL) for workflow control, acknowledgements and finally support for packet broadcast and multicasting. The development of an effective ‘space network’ will also require other characteristics from mobile ad-hoc networks (MANETs) including network mobility and scalability.

3. Communication Protocol: For ad-hoc networking capabilities, the IEEE 802.11 communication protocol standard is chosen. It is particularly important to note that there is a trade-off in power vs link properties which is outside the scope of this research. A range of 400 km using omni-directional antennas, a data rate of 1 Mb/s, and 1W RF output power is assumed from this point on in the research.

4. Picosatellite Network: For adaptable and redundant ground links, it is assumed that all picosatellite nodes are homogenous in sub-systems and a typical low-cost satellite system should be investigated for space sensor network applications. To account for satellite mobility due to current and future satellite constellations as well as the inclusion of orbital perturbations, there is a need for topology reconfiguration for mobile ad-hoc networks.

4.3 Summary

This chapter overviews how the drivers and technologies from the terrestrial distributed computing domain and current space systems can be used to help distributed satellite systems and their issues in existing intersatellite link and distributed operations. Requirements are then set out for a new distributed computing platform at node (single satellite) and network (multi-satellite) levels. A new hardware and software SoC design is proposed utilising a Java as a co-processor and Agent middleware for future networking operations to an existing space ESA proven SoC design. Main requirements are low ROM and RAM usage at a minimal overhead for new high and low data rate applications.
Chapter 5

5 Java Co-Processor System-on-a-Chip

This chapter elaborates on a proposed new system on a chip solution with a Java co-processor for wirelessly networked processors in the context of networked embedded systems. Sections 5.1 describes the main SoC design features. Section 5.2 investigates memory considerations regarding the footprint and shared memories/caches. Section 5.3 explains the design and integration as well as including work for hardware exceptions. Section 5.4 explores the implementation on a minimal FPGA platform for logic utilisation, timing, and power estimation results. Section 5.5 discusses the validation of the SoC with a combined bootloader for the additional Java co-processor and Section 5.6 overviews the design conclusions.

5.1 System-on-a-Chip Configuration

As previously described in Section 4.2, the proposed solution for the distributed mission computing requirements is a SoC system to provide node level and network level functionality. The node level functionality is achieved using the LEON3 processor [35]. The LEON3, originally designed by ESA, is a general purpose fully compliant SPARC V8 soft-core processor available with an extensive Gaisler IP library (GRLIB) from Gaisler Research [206]. The LEON3 processor, adopted by ESA as the main CPU for future on-board computers since the end of 2006 [207], is used for typical processing and ‘number crunching’ capabilities such as image compression. As previously described in Section 2.4.1, to accommodate an Agent computing environment, a JVM or JRE is required to provide a heterogeneous Java platform on which to realise an Agent middleware as a communication medium between various platforms.

As ‘Just in Time’ (JIT) compilation at runtime is far from time deterministic for embedded real-time critical systems, this work also takes advantage of the open-source JOP [47], described in Section 2.3.3.1.3, to enable real-time Java functionality on-board satellite systems. JOP is chosen as it is the smallest and fastest Java core to date. JOP is a RISC and stack based architecture used to execute Java bytecodes using microcode instructions. By utilising JOP in an FPGA along with the LEON3 processor, the benefits include:

- Moving software to hardware and reducing the middleware memory footprint with increased FPGA utilisation and increased speedup.
- Enabling Java applications, such as Internet networking applications or Agents, for real-time applications at JRE 1.1.8.

A block diagram can be seen in Figure 5-1 where JOP is integrated as a network and communications co-processor with the LEON3 using the AMBA2 AHB bus from ARM [208] for shared memory.

![Diagram of LEON3 and JOP Design](image)

**Figure 5-1. Overview of LEON3 and JOP Design**

A shared bus approach was chosen to provide fast, high-bandwidth and synchronised master-to-master communication for modifying registers in other AHB masters (e.g. once a network program is complete). It is envisioned that multiple Java co-processors could also be implemented for real-time thread-level parallel Java support using a shared memory. Table 5-1 overviews some of the key trade-offs.

<table>
<thead>
<tr>
<th>AHB</th>
<th>APB</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Basic Characteristics</strong></td>
<td>• High bandwidth &amp; burst transactions</td>
</tr>
<tr>
<td></td>
<td>• Synchronised with other AHB masters</td>
</tr>
<tr>
<td></td>
<td>• High complexity</td>
</tr>
<tr>
<td><strong>Design Specific Notes</strong></td>
<td>• Lower bandwidth</td>
</tr>
<tr>
<td></td>
<td>• Not synchronised with AHB masters</td>
</tr>
<tr>
<td></td>
<td>• Lower complexity</td>
</tr>
<tr>
<td></td>
<td>• Start &amp; stop Java processing</td>
</tr>
<tr>
<td></td>
<td>• Creates a dependency between AHB masters</td>
</tr>
<tr>
<td></td>
<td>• Closer to wireless front-end &amp; could reduce input latencies</td>
</tr>
<tr>
<td></td>
<td>• Incoming data requires additional management</td>
</tr>
</tbody>
</table>
Other alternatives would have been to implement the communications co-processor on 1) the APB bus or 2) a separate AHB bus for dedicated direct memory access capabilities. If implemented on the APB bus, the co-processor would operate at a lower bandwidth/speed and perform without AHB master synchronisation and therefore any core registers would require polling. But it would be easier and faster to implement the core on the APB and be physically closer to external wireless front-end components. If an additional AHB bus was used, there would be no bus contention allowing JOP access to memory any time it wishes. But conversely, the two AHB bus’s would require additional switching/arbitration logic.

5.2 Java Co-Processor Memory Considerations

To reflect this SoC design with an additional co-processor, the software design is layered as shown in Figure 5-2, where inter process communication (IPC), for exchanging data across multiple processes, can occur [210]. This can be achieved through software applications as messaging passing or via hardware schemes such as a synchronised shared memory or modification of registers within AHB masters.

![Figure 5-2. Hardware & Software Layer Design and IPC Methodologies](image_url)

For this design, IPC occurs in software where a combined registers and memory area approach is used to synchronise the shared memory. These registers and memory are accessible by AHB masters for when the Java co-processor completes routines/operations. Communication standards between the processors are required to retain coherency using a shared memory system and is provided by the robust AMBA bus scheme using register based buffering for read and write operations on the bus. A memory map is also used to separate each core’s designated addressable memory locations. The final software must have a low memory footprint (with operating environment and network stack) and still be real-time. A comparison of the memory footprint and functionality which looks at previous solutions to this problem can be found in Table 4-1, where there are three options considered:

1. A CORBA Middleware based implementation [56].
2. The standard Java libraries and software runtime used by PCs.
3. A new SoC design where the standard Java runtime is replaced in hardware and software is implemented with the CDC stack.

As described in Section 2.3.2.2, CDC and pjava are designed for the devices with intermittent network connections, slow processors and limited memory such as mobile phones, two-way pagers and personal data assistants (PDAs) – making them ideal to run in real-time on the JOP processor. Either 16-or 32-bit MPUs are required and a minimum of 128 kB to 512 kB of RAM for the Java platform implementation and associated applications. The full JRE 1.4 requires over 15 MB alone and is a major deterrent for using Java on embedded devices but dynamic class parsers are now available to help minimise the application to a very small size, discussed in Section 2.3.3.1.1. Table 5-2 shows combinations of operating systems and middleware discussed in Section 2.3 to make up distributed computing platforms for embedded networked systems.

<table>
<thead>
<tr>
<th>OSI Software Layer Method</th>
<th>Size (MB)</th>
<th>Real-time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Full Software using CORBA</td>
<td>1.739</td>
<td>√</td>
</tr>
<tr>
<td>(LEON3 + RTEMS, C++, ORB, 802.11 Driver, TCP/IP, Dyn. Lib.)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2. Full Software using Java</td>
<td>&gt;16.000</td>
<td>X</td>
</tr>
<tr>
<td>(LEON3 + RTEMS, JRE 1.4 Std. Lib, CDC 1.0)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3. Combined Hardware &amp; Software Design</td>
<td>1.106</td>
<td>√</td>
</tr>
<tr>
<td>(LEON3 + JOP, CDC 1.0 + JADE-LEAP)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

From Table 5-2, it can be seen that the third option offers the smallest memory footprint whilst retaining real-time functionality using the combined hardware and software design which uses CDC 1.0, JADE-LEAP and additional Agents has been reduced to 1.1 MB. Compared to the CORBA based implementation, the proposed system would reduce the footprint by 37 % and compared to the desktop Java solution, the memory footprint becomes too large and too slow for an embedded system.

5.2.1 Shared Memory and Caches

Multi-core system designs use shared memory for a fast form of IPC between the cores. Once the memory has been mapped, core synchronisation is required between the processes for storing or fetching data to and from shared memory, often called symmetric (shared memory) multiprocessing (SMP). The synchronisation is implemented using the open-source AMBA AHB Bus. The AMBA bus acts as the backbone in many SoC designs and is adopted here to provide
connection between the processor and peripheral cores, on-chip memories and off-chip external memory interfaces, shown in Figure 5-3.

The LEON3 system implements a standard data and instruction cache but JOP implements a 'stack' cache for data and 'method' cache for instructions which is designed for real-time worst case execution time (WCET) analysis. JOP's unique design for a hardware implementation of a JRE (at v 1.1) implements a simplified garbage collection (GC) model using the RTSJ specifications introduced in Section 2.3.3.1.2. This method schedules a GC thread for automatic memory management. JOP's 1 kB cache size was determined by a previous analysis of JRE 1.1 method lengths being 98% under than 1 kB [211] but can be increased if on-chip resources are available. A fault tolerant version of LEON3 also has a configurable cache and memory system designed to be tolerable to single event upsets (SEUs) or single event latchups (SELs) in the space environment with protected on-chip memories using triple modular redundancy (TMR), parity checking or duplication [212]. TMR is a fault tolerant design process where a design is replicated three times and a voting scheme is utilised to take the sets of data signals and produce 1 correct data signal.

The AMBA bus scheme here implements two types of bus: the advanced high-performance bus (AHB) and the advanced peripheral bus (APB). The AHB bus typically operates at 100 Mbps and is used for on-chip networking only whilst the APB bus operates at a much lower speed for I/O, typically 2 or 3 times less than the AHB bus. Bus access for IP core requests and the shared
memory access is handled using the AHB arbiter where AHB ‘Masters’ can request the bus which is then addressed, granted access, and locked for use before finally being released to the arbiter. Once the bus is locked, no other core can use the bus unless a *split-mode* of operation is implemented where the bandwidth is divided [213].

The LEON3 debug support unit (DSU) is a dedicated AHB slave interface for performing online debugging or instruction tracing of the processor and other masters on the AHB bus using register based buffers. The JTAG UART interface converts JTAG signals to AHB bus transfers and is used to program the FPGA bitstream as well as for debugging purposes. The memory controller on the APB bus is used as a generic slave for the host PROM, mapped I/O devices, and RAM devices [202].

### 5.3 Java Co-Processor Design Modifications

The advantage of using multiple cores on one configurable FPGA device is that a mix of technologies can be used, making the designs very versatile with differing memory systems and IP Cores but with high development time and costs. Therefore, the JOP core is integrated in such a way that it can be added to GRLIB for use in future projects. To add JOP as a non-heterogeneous Java based network processor, several issues need to be resolved:

1. **JOP and Bus Architecture**: JOP uses the SimpCon bus scheme [214] whilst the LEON3 uses the ARM AMBA2 bus. To add JOP as an AHB Master, an interface between the SimpCon and AMBA bus needs to be developed.

2. **Technology Primitives**: JOP can be successfully ported to any given hardware vendor by implementing two areas in the processor: the bytecode cache and stack which use RAM primitives. For FPGA based designs, Altera or Xilinx primitives are typically chosen.

3. **Exceptions**: JOP, like any JVM, has programming and network exceptions that could cause the processor to stall. These include class initialisation errors, stack overflow errors or System.exit network errors. These need to be handled to allow for restart of JOP under a different mode and for increased fault tolerance.

4. **Bootloader**: Both the LEON3 and JOP require off chip memory areas, typically in PROMs or FLASH, to hold the program and data code. These interfaces must be available to both cores so they can run separately from each other. As JOP avoids dynamic class loading, all required classes must be loaded on startup with known start addresses.
JOP has been integrated with an AHB Bus Master to SimpCon interface and APB configuration registers. The IP core is wrapped as one component for easy interfacing purposes at the top level and connects to AHB masters such as the LEON3 and memory via the AHB bus, shown in Figure 5-4. JOP itself operates 4 pipelined stages: *fetch*, bytecode *fetch* which is an additional translation from micro-code to bytecode, *decode*, and *execute*. The core itself uses additional interfaces to find initial start addresses and special pointer addresses and connections to external components are achieved using the memory core and the extension core. The *memory core* provides an interface between the main memory and the CPU whilst the *extension core* provides some extended functionalities including a multiplier unit, control signals for memory and I/O, and a multiplexer for read data to load to the top of the stack. An original I/O module has been replaced by AMBA interfaces [41], shown in Figure 5-4. The AMBA interfaces are an *AHB Master Interface* and an *APB Slave Interface* where both contain configuration information that is initially sent to the AHB arbiter. The AHB interface has an additional direct memory access (DMA) interface to perform read and write operations. The APB has some configurable control registers to set start and output addresses of the core as well as feedback for exceptions and debugging DMA signals. The LEON3, with both master and slave functions, is able to request (as a master) and serve (as a slave) to other cores on the AHB bus whereas JOP can only request as a master. The top level code and the complete wrapper code can be found in Appendix D.
5.3.1 Java Co-Processor Integration and Bus Architecture

As described in Sections 5.1 and 5.2, the SoC solution includes the following important cores implemented in the design, where the JOP co-processor is the major addition:

- LEON3 Central Processing Unit (CPU)
- AMBA Bus: AHB (high speed) & APB (peripheral)
- Debug Support Unit (DSU) & JTAG Debug UART
- New JOP CPU Wrapper with SimpCon to AMBA Interface
- ESA Memory Controller

The LEON3 is used as the main controller in the design providing node based functionality. The AHB Bus provides high speed communication between internal cores. The APB Bus provides communication to external peripherals, such as memory and other I/O. The DSU and debug UART provide useful debugging information from the FPGA device during development. The new JOP wrapper has the JOP CPU for network functionality, an AHB master to communicate with the LEON3 processor as well as other cores on the AHB bus, and an APB slave for configuration, registers, and I/O communication. The memory controller allows the synchronization between the SoC signals and external components.

Following the SimpCon Interface and AMBA2 AHB Interface specifications, a timing diagram is used to help develop the interface, shown in Figure 5-5. Both a read operation, where JOP requests data from a memory address, and write operation, where JOP writes data to a memory address is shown using AHB Master signals. The signals are as follows:

- **JOP SimpCon Signals** – addr = address, wr_data = write data, rd = read, wr = write, rd_data = read data, and rdy_cnt = ready counter (usually linked to external memory).

- **AMBA AHB Signals** – ahb_raddr = AHB read address, ahb_rdata = AHB read data, ahb_waddr = AHB write address, and ahb_wdata = AHB write data.

- **DMA Signals** – dmao_active = direct memory access output active, dmao_ready = direct memory access output ready, dmao_rdata = direct memory access output read data, dmai_addr = direct memory access input address, and dmai_wdata = direct memory access input = write data.
Figure 5-5. Timing Diagram from SimpCon and AMBA2 AHB Interface

The red and blue arrows denote the data flow going to and from the AHB DMAI/O bus. The read operation passes JOPs address to the AHB registers and, when the memory is active and ready, reads the data from the address back to AHB registers and JOP’s interface. The write operation is the same as the read operation except JOP passes both data and an address to be written to memory. When implementing the design, it is important to note that the interface must use a ‘load’ state for setting address registers as well as an ‘idle’ state when the system is reset. The use of rdy_cnt in the SimpCon interface is related to the SRAM previously implemented. Here, SRAM is not used and this signal is redundant and set to ‘0’ (no waiting). The rdy_cnt can be used to stall the processor pipeline for granted AHB access when it is set to ‘1’. The AHB and SimpCon successfully interfaces together and reduces the number of clock cycles required for read/write operations by up to 4 clock cycles. No other on-chip bus architectures were studied as it is out of the scope of this research.
ModelSim SE 6.1a [215] is used to simulate interfacing the JOP SimpCon interface, the AMBA APB interface to PROM, and the AHB interface to RAM. A simulation screenshot is found below of the read operation is found in Figure 5-6.

Figure 5-6. Behavioural Simulation of Reset, Load and Read from JOP to Memory

At the beginning of the simulation in Figure 5-6, a reset signal is sent to the JOP IP core whereby it enters the load state. In this state, the state machine waits for the SimpCon interface from the JOP CPU to request either a ‘read’ (where the read line goes high and the read address register is loaded) or a ‘write’ (where the write line goes high and the write address and data registers are loaded). When performing a read operation, the address is offset by the APB bus configuration lines so that the correct memory address is used for reading JOP’s footprint (stored in a bootloader discussed later in Section 5.5.1) and retrieving the initial bytecodes, in this case to the address 0d:cc:001. The operation waits until the direct memory access lines are ready, then takes data from the off-chip memory via the direct memory access interface, and loads AHB and JOP registers to complete the operation. This sequence demonstrates how the JOP IP core is first used to integrate the cores with the external memory interfaces to retrieve data using the AHB and APB buses.
Further behavioural simulation of the JOP CPU are found in Appendix E, showing the first characters appearing externally after 7940 ns and a greater detail screenshot of the bytecodes being executed by the processor.

5.3.2 Technology Primitives

To implement the stack and bytecode cache, the correct technology primitives are required for a specific FPGA vendor. For Xilinx FPGAs, four ‘RAMB16_S9_S9’ modules are used for the stack and one ‘RAMB16_S9_S36’ for the bytecode cache. Upgrading to larger BRAM resources found in larger FPGAs could also speed up the stack and bytecode fetch pipeline related processes.

5.3.3 Hardware Exception Handling

Exception handling has been problematic in fault tolerant systems. Some relevant examples are discussed by H. Hecht [216] including examples of catastrophic failures with an Ariane-5 launcher and the Mars Polar Lander. For a SoC design, there are two types of failures: global failure, where many functional areas of the device are affected requiring a device reset, and a recoverable failure, where processes in hardware and software can cause functional errors in specific areas of the device. To deal with these recoverable errors, there are several main hardware and software exceptions that could occur in the Java processor to be concerned with.

Hardware exceptions include:

1. Stack Overflow – where the stack becomes full, typically due to a large number of classes
2. Null Pointer – an address which has elements undefined or is out of the memory scope
3. Array Out of Bounds – access to an array or array element which may not be accessible

Whilst software exceptions include:

4. Network Exceptions – timing constraints not met or unhandled protocol exceptions
5. Application Specific Exceptions

Each of the hardware exceptions typically results in the stalling of the processor and a hard reset is required which are typically due to overloading of the processor or corrupt software. In this case an automatic hardware reset is generated from a separate entity shown in Figure 5-7. Software exceptions can occur due to poor network connectivity, application specific exceptions or programming errors. To overcome these software problems, a supervisory ‘Instance Manager’ thread is used (described in Section 6.2.1) by opening applications as threads and performing soft resets.
when enabled =>
    if enable = '1' then reset_ft <= reset; next_state <= idle;
    else reset_ft <= reset; next_state <= enabled;
    end if;
when idle =>
    if enable = '1' then
        if spov = '1' or ab = '1' or np = '1' then reset_ft <= not reset; next_state <= error;
        else reset_ft <= reset; next_state <= idle;
        end if;
    else reset_ft <= reset; next_state <= enabled;
    end if;
when error =>
    if count = 5 then count := 0; next_state <= idle;
    else reset_ft <= not reset; count := count + 1; next_state <= error;
    end if;

Figure 5-7. Hardware Exception Reset Loop

In the case shown in Figure 5-7, any hardware exception will all cause a reset (held for 5 clock cycles) so the processor can be brought back online in the shortest time possible. If an exception occurs during processing or utilising shared memory, the Java processor IP core wrapper has additional registers which can be set to warn other AHB masters there has been a recent error. JOP's method and data caches are initially filled on startup and would not to be reloaded for approximately 30 read operations.

5.4 SoC Design Simulation Results

As discussed in Section 4.2, the final SoC configuration aims to have minimal overhead when adding the additional Java co-processor. This will also reduce the overhead to the picosatellite testbed and the final power requirement needed. As such, the existing JOP I/O module has been removed leaving just the CPU (refer to Section 2.3.3.1.3, Figure 2-8).

Xilinx ISE 10.1 [217] is used to synthesise, map and place & route the SoC to produce a bitstream for downloading to a Spartan-3 1500 FPGA device [218], discussed in greater depth in Section 5.5. Figure 5-8 shows the final layout on the Spartan-3 1500 FPGA where some main components are displayed. Namely, LEON3 (red), AHB Bus (blue), APB Bus (green), and JOP (yellow).
Figure 5-8. Routed FPGA Design

From the layout in Figure 5-8, JOP utilises existing logic area in the FPGA and is optimally placed so that it has better access to APB and memory, the LEON3 processor, and the AHB bus. Alternative layout design strategies for the layout were evaluated for 1) timing and 2) minimal resource performance to find there is negligible difference between any one strategy.

5.4.1 Logic Utilisation Results

On-chip resources are scarce and the overhead for adding the Java processor must be taken into consideration as it will affect the final power consumption. Table 5-3 shows the logic utilisation for the Spartan-3 1500 FPGA with an increase of 24% slices used and a significant 44% increase in BRAMs used. This work also confirms that the additional co-processor added is small enough to fit in a relatively small FPGA and allow additional functionality to be implemented using the remaining logic slices. When realising the caches for both processors, more block RAMs are filled in the Spartan-3 1500 FPGA meaning that allowing TMR or protecting the memory areas from SEU/SEEes would not be possible and a larger device would be required.
Table 5-3. Device Utilisation Comparison Summary on the Spartan-3 1500 FPGA

<table>
<thead>
<tr>
<th>Logic Utilisation</th>
<th>Available</th>
<th>LEON3</th>
<th>LEON3 + JOP</th>
<th>Increase (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Number Slice Registers</td>
<td>26,624</td>
<td>3,579</td>
<td>4,783</td>
<td>34</td>
</tr>
<tr>
<td>Total Number of 4-Input LUTs</td>
<td>26,624</td>
<td>11,872</td>
<td>14,666</td>
<td>24</td>
</tr>
<tr>
<td>Logic Distribution</td>
<td>13,312</td>
<td>6,855</td>
<td>8,263</td>
<td>21</td>
</tr>
<tr>
<td>No. of Bonded IOB Flip-Flops</td>
<td>333</td>
<td>162</td>
<td>162</td>
<td>0</td>
</tr>
<tr>
<td>No. of RAMB16s</td>
<td>32</td>
<td>16</td>
<td>23</td>
<td>44</td>
</tr>
<tr>
<td>No. of MUX18x18s</td>
<td>32</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>No. of BUFGMUXs</td>
<td>8</td>
<td>4</td>
<td>6</td>
<td>50</td>
</tr>
<tr>
<td>No. of DCMs</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>No. of BSCANs</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 5-4 presents the difference in on-chip LC utilisation between a LEON3, the JOP with an AMBA interface and finally the combined LEON3 and JOP with AMBA interfaces showing that adding JOP uses an additional 2794 LCs. This 24% increase in LCs will also increase the power consumed as more registers and logic is being used but enable network processing capabilities using the new IP core.

Table 5-4. Summary of Integration Utilisation on GR-XC3C-1500 FPGA Board

<table>
<thead>
<tr>
<th>Component</th>
<th>LCs Used</th>
</tr>
</thead>
<tbody>
<tr>
<td>LEON3 CPU + AMBA Bus System</td>
<td>11,872</td>
</tr>
<tr>
<td>JOP CPU + AMBA Interface</td>
<td>3,252</td>
</tr>
<tr>
<td>LEON3 + JOP + AMBA Bus</td>
<td>14,666</td>
</tr>
</tbody>
</table>

5.4.2 Static Timing Results

Implementation results of this design give a maximum frequency of 58.023 MHz when targeting the Spartan-3 1500 FPGA. The arbitration and shared memory scheme using the AHB bus between the LEON3, JOP, and other cores suggests that the maximum speed of the whole design will also be reduced. One of the largest latencies that can reduce the overall maximum frequency is the routing of clock signals and, in this SoC design, they must pass through up to 23 levels of logic in some component instances. To alleviate this problem, multiple AHB clock signals can be used to allow the processor to reach higher performance, often called dynamic clock switching or through split-mode operation [196].

Three key signal domains to check were:
1. LEON3 CPU to Memory Controller  
   (8.444 ns through logic, 13.099 ns through routing, 16 levels of logic = 58.023 MHz)

2. LEON3 CPU to Ethernet 10/100 Transmitter  
   (5.519 ns through logic, 7.031 ns through routing, 8 levels of logic = 79.679 MHz)

3. JOP Wrapper Registers to Memory Controller (read and write)  
   (3.678 ns through logic, 7.481 ns through routing, 5 levels of logic = 89.613 MHz)

The key constraint these signals must meet is 40 MHz, the operating speed of the chosen FPGA board (described in Section 5.5) and the minimum frequency for IEEE 802.11 b/g, set in Section 4.1. No optimisation was attempted on signals 1 and 2 but checked so the frequency constraint is met. A trade-off between the area and speed is usually required, but these results show that the new SoC design can operate at higher frequencies and therefore does not impede the maximum speed of the original system. To find out how fast this design could operate, a Virtex-5 LX50 was chosen based from the Gaisler Virtex-5 PCI FPGA Board [219]. Preliminary timing results showed that the complete design could operate at 180.976 MHz with a device utilisation was only at 17 % of total LCs, which exceeds the previous JOP speeds of 100 MHz.

5.4.3 Power Estimation Results

Power is measured as both static power, typically based on the transistor count, and dynamic power, based on the switching or toggle rate of the transistors.

Power consumption of three configurations are estimated in Xilinx’s XPower [220] including the LEON3, LEON3 plus JOP, and JOP for each FPGA design excluding any off-chip components. Input variables include:

- Ambient Temperature of 10 °C.
- Zero Airflow.
- Spartan-3 package thermal resistance (ΘJA) °C/W of 20.4 and maximum junction temperature of 93.6 °C.

For a standard toggle rate, which is how often the logic output of a design changes with respect to 100 clock signals, of 12.5 [221] and operating at 40 MHz, the complete SoC design is predicted to consumes 291 mW (149 mW quiescent and 142 mW dynamic) with negligible overhead difference compared to the other separate designs of a single LEON3 or a single JOP configuration.

Figure 5-9 also compares the power consumed at higher toggle rates between the LEON3, LEON3 + JOP, and JOP alone to show that there is an estimated power increase of 29 mW.
comparing the new SoC design to a single JOP configuration and 13 mW to a single LEON3 configuration.

Further analysis of the design looks at the use of functionality, checking BRAM memory usage, and the power consumption variance when passing through a LEO thermal cycle (approximately 60 minutes in sunlight and 30 minutes in darkness). These are found in Figures 5-10 and 5-11 respectively.
Figure 5-10 shows that the main issues in this SoC design are in relation to static powers and I/O power (to the memory system) and not the additional on-chip BRAM usage. In this design, the power for on-chip BRAM is much smaller than the number of active I/O lines required. Figure 5-11 also shows that despite the extreme thermal cycle, the power consumption is moderately stable. These power readings estimates are used for the on-board testbed, discussed in Section 7.3.3. Adding triple modular redundancy (TMR) is often an important feature to make this design flight ready but as the Spartan-3 device chosen has high utilisation for this design, triple modular redundancy could not be implemented without changing to a larger device. Adding TMR would ensure higher reliability in space but was not considered for these power estimation results.

5.5 SoC Design Validation

The board used for experimentation was the GR-XC3S-1500 FPGA Development Board from Pender [222], shown in Figure 5-12. It has 18 MB PROM, 64 MB SDRAM, has a clock speed of 50 MHz, and requires 3.3V for I/O and 1.2V at the core.
The targeted development board was configured with the SoC design bitstream and debugged using a JTAG cable. As the core is designed to be attached as a ‘plug & play’ (PnP) device, GRMON [223] is used to check for the JOP IP-core. GRMON is an online debugger for the LEON3 processor using the debug support unit (DSU). The debug UART from Figure 5-13 (a) is used to check the AMBA bus configuration. Figure 5-13 (b) shows JOP as an attached core as ‘unknown device’ at the address 04:cc:001 on the AHB bus and 0:cc:001 on the APB bus. The core is identified as ‘unknown’ because the codes developed in this research have not been added to the GRMON and GRLIB device listings by Gaisler.

5.5.1 Bootloader Design and Implementation

The bootloader on the target board has a secondary requirement, that there is only 1 PROM that must hold bootloaders for both processors. In typical multi-processor systems, either the processor cores are a) each processor has its own PROM for a or b) bootloader identical or homogenous and the same bootloader can be used for N processor cores. This is particularly problematic when implementing the Java co-processor core as depicted in Figure 5-14, showing the issue where this design needs a different approach and solution.
Figure 5-14. a) Processors with Separate Memory Systems & b) with SMP systems

Figure 5-14 shows two common configurations for implementing multi-processor architectures but a new methodology is required to combine both bootloaders in the same PROM.

The LEON3 Processors applications are developed using Gaisler RTEMS 4.6 Tools [224] and JOP applications are developed using the JOP Tools [225]. Table 5-5 describes the files and processes for application development for each processor.

Table 5-5. Application Creation Files and Processes for LEON3 RTEMS 4.6 & JOP

<table>
<thead>
<tr>
<th>Initial Files</th>
<th>LEON3 RTEMS 4.6</th>
<th>JOP</th>
</tr>
</thead>
<tbody>
<tr>
<td>C, C++, header files</td>
<td>Compile into 1 of more object files using sparc-rtems-gcc.exe</td>
<td>Java, Jar files</td>
</tr>
<tr>
<td></td>
<td>Create final image using sparc-rtems-mkprom.exe</td>
<td>Assembles microcoded JVM &amp; produce memory initialisation files using Jopa</td>
</tr>
<tr>
<td>Output Format</td>
<td>prom.out (32-bit)</td>
<td>Link the java application &amp; converts the class information to a .jop file using JOPizer</td>
</tr>
<tr>
<td></td>
<td>prom.jop (32-bit)</td>
<td></td>
</tr>
</tbody>
</table>

The JOP Java application is compiled first to bytecode, then to microcode, and finally linking with class files completes with a .jop file. To overcome the shared memory architecture of this design, compilation of each core’s application must be stored together in the same footprint image. The processes and tools are illustrated in Figure 5-15.
Figure 5-15. Combination Process for LEON3 and JOP Applications

Figure 5-15 shows that standard open source tools are used to compile each application. JOP need to utilise SRECORD tools 1.49 [226] or similar object copy programs once again to convert from a binary format to a .srec format. They can then be concatenated together at differing start addresses to avoid confusion of each processor core’s start address. The LEON3 application has its code (.text segment) typically stored in a PROM at 0x00000000, and data (.data and .bss) in RAM at 0x40000000. At start-up, the .data segment is copied from the PROM to the RAM, linked to start from address 0x0. The data segment is by default linked at 0x40000000 also but can be changed by giving offset arguments which is the technique used to set JOP’s application. JOP’s application is aimed at starting at address 0x41000000 and outputting to 0x42000000, away from the LEON3 memory area. These start addresses can be set in a C program by the LEON3 or hard-coded in the JOP IP core wrapper component.

The initial data segments in Figure 5-15 are confirmed using srec_info.exe and demonstrated in the ModelSim simulation found in Figure 5-16.
In Figure 5-16, the APB control registers are set to the previously mentioned addresses and the AHB signals change dependent on the AHB arbiter and JOP SimpCon interfaces. It can be seen that there is an initialisation phase where there are many unknown values of JOP’s stacks used for mathematical functionality\(^1\) due to a lack of program input in simulation. After this initialisation phase, the read signal goes high and the correct address signals are set, awaiting for direct memory access. As soon as the arbiter receives the request to use the bus (ahbmo.hreq not shown here), memory access is granted.

### 5.6 Summary

This chapter describes the technical process of integrating the JOP processor to the AMBA bus to operate as a Java co-processor in parallel with the LEON3 processor. This aims to provide a real-time Java runtime environment (JRE 1.1) to an existing flight ready system-on-a-chip solution in

---

\(^1\) Tos = top of stack, nos = next of stack. When a number, integer or variable ‘A’ is called, nos = tos and tos = A. Signed values are typically put on the nos, unsigned values on tos are for finals.
Chapter 5. Java Co-Processor System-on-a-Chip

A FPGA for future Java networking applications. By including a Java processor to the AHB bus, synchronous high bandwidth and burst operations are achieved in a shared memory environment with other AHB masters. Alternatives to this implementation using a second dedicated AHB bus and as an APB slave are also discussed to be viable alternatives of this architecture. Interprocess communication is implemented using simple register polling/modification.

A comparison of existing distributed computing technologies shows that this SoC design has a smaller software footprint than existing solutions. Implementation has been carried out addressing issues such as technology primitives, interfacing, hardware exceptions, and bootloading. Post place and route simulations confirm correct interfacing between JOP's SimpCon and the AMBA AHB/ APB buses, existing timing constraints were still met, and software can be loaded using a combined bootloader for both the LEON3 and JOP processors. The evaluation of this new design finds that the LEON3, JOP co-processor and on-chip bus fits in the small Spartan-3 FPGA device at an estimated power consumption of 291 mW and a 24% LC overhead and 44% BRAM overhead to the existing LEON3 SoC design. The design is also resilient to temperature changes that could occur in LEO but a larger device would need to be targeted to add triple modular redundancy and ensure flight readiness.

It is envisioned that this design could be used towards future multi-Java processors for real-time thread-level parallel Java support as a plug & play IP core in the Gaisler library.
Chapter 6

6 Agent Middleware and Application Development

This chapter describes the development of the Agent middleware and presents a case study to confirm functionalities. It aims to complete the distributed satellite system requirements set in this research and enable network level applications. Section 6.1 critically compares and analyses the major existing Agent middleware solutions using a novel method probe to check the online memory consumption of a standard Agent testbench application. The footprint, RAM usage, conformity to the proposed Java co-processor JRE 1.1.8 requirement, and timing performance are also discussed in detail and a final software stack is chosen based on JADE-LEAP-pjava. Section 6.2 explains the software components as a service orientated architecture and some of the common functionality now available to the platform including an instance manager, migration, inter-process communication and parallel threading. Section 6.3 presents simulations of computing applications required towards mobile ad-hoc networks that take advantage of the Agent paradigm.

6.1 Comparison of Agent Middleware Systems

As described in Section 2.4, there are various Agent middleware options available to the research community with the majority using derivatives of JADE or FIPA-OS development environments. Each Agent platform has dependencies based on a particular Java revision environment (JRE). For example, JADE can be implemented based on JRE 1.4 or as JADE-LEAP using JRE 1.2. JADE-LEAP can then be configured under J2ME, PersonalJava (or pjava) now superseded by the Connection Devices Configuration (CDC Spec.) or the Mobile Information Device Profile (MIDP) stack which uses the Connection Limited Device Configuration (CLDC Spec.). FIPA-OS is also considered along with Micro FIPA-OS, targeted for mobile phones.

To perform a complete analysis, all open-source FIPA conforming Agent middleware systems are considered in this project for a comparison of footprint, RAM usage and conformity for future applications. An initial comparison in Table 6-1 compares each Agent middleware against requirements in Section 4.2. This aims to ensure that the chosen platform conforms to the Java
processor JOP's runtime environment constraint of JRE 1.1.8 to operate the Agent middleware, described in Sections 2.3.3.1.3 and 4.4. Crucial factors compared include the JRE version, target platforms, and additional advantages and disadvantages. Notes in grey are limitations of that configuration towards an embedded Agent middleware. Agent functionality can be broken into key functions: mobility (already discussed in Section 2.4), cloning, and backup functionality. Cloning is the ability to replicate and copy the entire Agent whilst backup functionality is the ability to take over platform Agent management operations should any problem arise.

Table 6-1. Agent Middleware Comparison

<table>
<thead>
<tr>
<th>Configuration</th>
<th>JRE Version</th>
<th>Target</th>
<th>Advantages/ Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>JADE 3.5</td>
<td>1.4</td>
<td>Target is Desktop PCs</td>
<td>Desktop Agent functionality</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Community support</td>
</tr>
<tr>
<td>JADE-LEAP</td>
<td>1.2.0 (J2SE)</td>
<td>Target is Mobile Phones/ PDAs</td>
<td>Embedded Agent Functionality</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Community support</td>
</tr>
<tr>
<td>JADE-LEAP-pjava</td>
<td>1.1.8 (CDC)</td>
<td>Target is Mobile Phones/ PDAs</td>
<td>Embedded Agent Functionality</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>No Backup functionality</td>
</tr>
<tr>
<td>JADE-LEAP-midp</td>
<td>1.1.8 (CLDC)</td>
<td>Target is Mobile Phones/ PDAs</td>
<td>Embedded Agent Functionality</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>No Backup functionality</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>No mobility</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>No cloning</td>
</tr>
<tr>
<td>FIPA-OS</td>
<td>1.3.0 (J2SE)</td>
<td>Target is Desktop PCs</td>
<td>Desktop Agent functionality</td>
</tr>
<tr>
<td>MicroFIPA-OS</td>
<td>1.1.8</td>
<td>Target is Mobile Phones/ PDAs</td>
<td>Embedded Agent Functionality</td>
</tr>
</tbody>
</table>

Table 6-1 shows that 3 of the 6 investigated state-of-the-art Agent middleware platforms require a higher JRE revision than JOP's JRE 1.1.8 constraint and are therefore unsuitable for hard real-time applications. They could however be used for soft real-time application experimentation on desktops or mobile phones. The CDC configuration found in JADE-LEAP-pjava, JADE-LEAP-midp and MicroFIPA-OS conforms to JOP JRE 1.1.8 requirement and would allow Agent mobility and cloning functionalities whilst still reducing the RAM memory requirements from J2ME. Targeting CDC offers the most relevant functionality for intermittent wireless networking whilst still retaining distributed operations at a minimum footprint – ideal for a distributed satellite system.

6.1.1 Test Methodology

Using Eclipse's Probekit from the Test & Performance Tools Platform (TPTP) Project [227], code was developed to make discrete measurements about a target Agent platform whenever a method
is entered which is referred to as a *method probe*. In this case, random access memory (RAM) measurements are taken and the result sent to a log file along with the class name and method name. This will aid in identifying blocks of code that cause increased memory usage. The probe’s code is found in Figure 6-1.

```java
import java.io.*;

class myprobe {  // Class for probe ssc.probe
    public static class Probe_0 {
        public static void _entry(String *className*, *aclassName0*,
            String *methodName*, *amethodName1*) {
            try {  // Find the Memory Usage
                Runtime runt = Runtime.getRuntime();
                long freeMemory = runt.freeMemory();
                long totalMemory = runt.totalMemory();
                long s = totalMemory - freeMemory;
                boolean append = true;  // Printout to File
                PrintStream out = new PrintStream(new FileOutputStream("c:/probe.txt", append));
                out.println(aclassName0 + " : " + amethodName1 + " : " + s);
                out.close();
            } catch (IOException e) {
            }
        }
    }
}
```

**Figure 6-1. Method Probe for RAM Measurements**

Figure 6-1 shows the method probe code which first finds the class and method, gets a handle on the runtime environment to check the memory used (total memory minus free memory) and finally print out to a log file. This will detect the memory change between each method used in the middleware to highlight areas that could be improved.

### 6.1.2 Open Source Agent Middleware Performance Experiments and Results

To functionally test the platforms shown in Table 6-1, a basic testbench startup application is loaded as the following:

- Any graphical user interface is removed and the middleware startup is automated to include the `System.currentTimeMillis()` method from JRE 1.0 to find the startup time.

- Load the Agent Management Service (AMS) and Directory Facilitator (DF) services on the platform. These are two core services used to register and find Agents across other platforms.

- Load 1 ‘HelloWorld’ Agent and complete registration with AMS and DF. Print the time to mark the ending of the testbench and exit application once complete.
Each Agent platform is FIPA compliant and operates a standard set of Agent functionality. The above testbench is used to find the ROM footprint, RAM usage, startup times, and porting capability to the Java processor, JOP. The method probe examines the number of methods called and the RAM usage under a minimum required JRE. The testbench results are found in Table 6-2.

Table 6-2. Agent Platforms – Un-optimised Functional Results

<table>
<thead>
<tr>
<th>Agent Middleware Configuration</th>
<th>Footprint (kB)</th>
<th>RAM Usage Min, Max, Mean Ave. (kB)</th>
<th>Time to Load (ms)</th>
<th>JOP Porting Capability</th>
</tr>
</thead>
<tbody>
<tr>
<td>JADE 3.5</td>
<td>866 (jade.jar)</td>
<td>180</td>
<td>8750</td>
<td>Not possible due to incorrect JRE</td>
</tr>
<tr>
<td></td>
<td>667 (jadeTools.jar)</td>
<td>1349</td>
<td>7547</td>
<td>Target: JRE 1.4</td>
</tr>
<tr>
<td></td>
<td>43 (iiop.jar)</td>
<td>869</td>
<td>7344</td>
<td></td>
</tr>
<tr>
<td></td>
<td>24 (http.jar)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>46 (common-codecs-1.3.jar)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3 (sscAgents.jar)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Total = 1649</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JADE-LEAP</td>
<td>2325 (JadeLeap.jar)</td>
<td>201</td>
<td>5172</td>
<td>Not possible due to incorrect JRE</td>
</tr>
<tr>
<td></td>
<td>46 (common-codecs-1.3.jar)</td>
<td>1214</td>
<td>5188</td>
<td>Target: J2SE-1.5</td>
</tr>
<tr>
<td></td>
<td>Total = 2369</td>
<td>766</td>
<td>5172</td>
<td></td>
</tr>
<tr>
<td>JADE-LEAP-pjava</td>
<td>1104 (JadeLeap.jar)</td>
<td>177</td>
<td>329</td>
<td>Possible</td>
</tr>
<tr>
<td></td>
<td>46 (common-codecs-1.3.jar)</td>
<td>873</td>
<td>328</td>
<td>Target: CDC-1.0/ Foundation-1.0</td>
</tr>
<tr>
<td></td>
<td>Total = 1150</td>
<td>531</td>
<td>328</td>
<td></td>
</tr>
<tr>
<td>JADE-LEAP-midp</td>
<td>637 (JadeLeap.jar)</td>
<td>86</td>
<td>158²</td>
<td>Not possible due to .JAD requirement (user required)</td>
</tr>
<tr>
<td></td>
<td>1 (jade-leap.jad)</td>
<td>535</td>
<td>153</td>
<td>Target: MIDP-2.0/ CLDC-1.0</td>
</tr>
<tr>
<td></td>
<td>Total = 638</td>
<td>313</td>
<td>153</td>
<td></td>
</tr>
<tr>
<td>FIPA-OS</td>
<td>855 (FIPA_Osv2_1_0.jar)</td>
<td>198</td>
<td>1937</td>
<td>Not possible due to incorrect JRE</td>
</tr>
<tr>
<td></td>
<td>8 (FIPAOSClassLoader.jar)</td>
<td>2261</td>
<td>2125</td>
<td>Target: JRE 1.3</td>
</tr>
<tr>
<td></td>
<td>52 (databinding.jar)</td>
<td>1599</td>
<td>1953</td>
<td></td>
</tr>
<tr>
<td></td>
<td>77 (jdom.jar)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>38 (SiRPAC-1.14.jar)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1451 (xerces.jar)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>66 (zeus.jar)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Total = 2547</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MicroFIPA-OS</td>
<td>1268 (mfos.jar)</td>
<td>245</td>
<td>812</td>
<td>Possible</td>
</tr>
<tr>
<td></td>
<td>23 (aelfred.jar)</td>
<td>1125</td>
<td>766</td>
<td>Target: CDC-1.0/ Foundation-1.0</td>
</tr>
<tr>
<td></td>
<td>254 (collections.jar)</td>
<td>805</td>
<td>781</td>
<td></td>
</tr>
<tr>
<td></td>
<td>9 (sax.jar)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Total = 1554</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

² MIDP here is on a Mobile Phone (Sony Ericsson W580i). The application tries to open a connection in ‘split execution’ mode and errors out due to a lack of WiFi support. This explains the low memory usage and low startup time.
Table 6-2 displays all the possible Agent middleware platform configurations and compares the middleware footprints, the RAM usage, their times to load, and the JOP porting capability where grey notes denotes incompatibility. The middleware footprints range between 638 kB to 2.5 MB which is comparable to existing work in Section 2.3.2.1 at 851 kB to 1 MB, the experimental work in [28] at 1.7 MB. The RAM usage ranges from 86 kB to 1.6 MB, giving an indicator of how much room is required if designing a board. Both JADE-LEAP-pjava and Micro FIPA-OS both can consistently lower RAM usage than the other desktop targeted configurations. Both the footprint and RAM usage should be kept to a minimum as it will increase the power consumption as described in Section 4.1.

The startup time is also very important where JIT compilation and their respective startup times are measured to be between 328 ms to over 8.7 seconds. For this experiment, the delayed startup is typically due to the JIT compiler searching the JRE libraries [228] for the required class/method. The startup times are initially higher but fall to a consistent memory usage value after 3 or 4 runs of the testbench as the classloader caches frequently used methods. JADE-LEAP-pjava is clearly the fastest to startup due to the minimal library used when aimed at an embedded system. Other configurations including JADE 3.5, JADE-LEAP-j2se, and FIPA-OS all took seconds rather than milliseconds. On the satellite, the startup time would need to first produce a reset and load the bootloader before finally loading the middleware and Agents.

When identifying the porting requirement for JOP, there are only two possible configurations: JADE-LEAP-pjava and Micro FIPA-OS. Unfortunately for JADE-LEAP-midp, the main configuration uses a .jad file to startup the JADE-LEAP.jar on the embedded device, in this case a mobile phone. The MIDP configuration can only be ran as a split-mode container, running the FrontEnd (the mobile phone), with a remote server BackEnd (typically a PC) [229]. This is unsuitable for two reasons:

1. Despite the mobile phone FrontEnd having a fast bootstrap time due to the offloading of network functionality on the BackEnd PC, the split-mode requires a permanent connection between the FrontEnd and BackEnd. Working in a mobile ad-hoc environment deems this unsuitable.

2. At this time, the low computational capability of the FrontEnd running on an embedded device is also not sufficient to allow for complex Agent computing operations without the input from a user.

### 6.1.3 Critical Analysis of Existing Agent Middleware

The method probe was used to obtain RAM usage results for each platform as well as the total number of methods required to complete the testbench. The results are displayed in Figures 6-2
and 6-3 (with and without FIPA-OS). The x axis is the number of probe measurements taken, i.e. the number of methods used to complete the testbench and the y axis shows the RAM usage in kilobytes (kB).

**Figure 6-2. Comparison of RAM Usage for Platform Startup using Probe**

**Figure 6-3. Comparison of RAM Usage for Platform Startup using Probe (without FIPA-OS)**

From the results in Figure 6-2, it is clearly seen that the pjava configuration under CDC has the lowest memory consumption and that FIPA-OS has the highest. The FIPA-OS configuration
initiates a Task Manager component which uses ACL messages, described in Section 2.4.2, between the AMS and DF Agents (and other Agents loaded) to ensure that synchronicity and timing constraints are met which adds significant overhead to this middleware configuration being able to timely complete the testbench application. In Figures 6-2 and 6-3, the memory consumption rises but this levels off immediately after the testbench finishes loading. By looking at the number of probe entries to find the number of method calls in Figure 6-3, there is a clear minimisation in RAM usage and method calls from each Agent platform as the target becomes more embedded to complete the testbench. The MicroFIPA-OS configuration, despite using more memory, makes almost half as many method calls to complete the testbench thus requiring a smaller stack for static class loading in this SoC design which also reduces the power consumption.

6.1.4 Final Software Stack Configuration

From Section 4.1, a low footprint and RAM memory usage is highly desirable and, from Section 6.1.3, reduced class loading is also highly desirable for running in an embedded environment, JADE-LEAP-pjava has been chosen as the final software stack configuration, see Figure 6-4.
This software configuration offers:

1. The CDC stack of standard Java methods usable for networking applications at JRE 1.1.8 either offered by JOP in hardware or in open-source software for emulation.
2. The lowest memory consumption when compared to other competing systems.
3. Agent functionality through JADE-LEAP with cloning capabilities.

This software stack has since been taken forward for development and its footprint reduced to 305 kB using ProGuard [230], an open source Java software shrinking tool, for shrinking, optimisation, and obfuscation, keeping only the required core classes for the middleware operation and communication. Shrinking analyses the main application and removes unused classes, fields, and methods. Optimisations include removing debug and logging codes, making classes static and final, and reduces variable allocation. These are mostly coding optimisations. Obfuscation is the replacement of naming in the classes, fields, and methods with simple characters and values. Despite being used to ensure code cannot be reserve engineered for greater security when the final Agent middleware is deployed, it also compacts the code.

When compared to previous middleware solutions in Sections 2.3 and 5.1.2, this is a reduction of 72% of the existing JADE-LEAP-pjava solution and 64% of a CORBA solution making it a very small Agent middleware solution for networked embedded systems.

6.2 Agent Based Service Orientated Architecture

In the development environment for JADE-LEAP, a *container* is a predefined area of memory that the middleware has available to different types of Agents. Here, non-mobile and mobile services are disseminated into two memory areas. Standard services include:

- Agent Management System (AMS) which exerts control over access and use of the platform as well as life-cycles, directory of Agent identifiers (AID) provision and Agent states.
- The Directory Facilitator (DF) provides ‘yellow pages’ for Agents registration and search services across platforms.

JADE-LEAP automatically loads kernel-level services, mobility, and event notification. Other services can be developed and added such as UDP communications, Agent security, and Agent persistence. JADE-LEAP typically uses permanent TCP connections to detect monitored remote platforms, provide message passing communications, and any other middleware service. This design also utilises a UDP mechanism for faster scalable ‘store and forward’ communication and topic based communication using multicast messaging (addressed by their AID). The final
middleware and service components are shown in Figure 6-5 wrapped around using the Instance Manager thread, described in Section 6.2.1. This new software architecture is called JADE Fault Tolerant (JADE-FT).

![Diagram of JADE-FT Middleware and Service Components]

**Figure 6-5. JADE-FT Middleware and Service Components**

JADE-FT is fault tolerant at the software level using the Instance Manager to manage software exceptions and network profiles in the JADE-LEAP Agent middleware (discussed further in Section 6.2.1). Other than the AMS and DF, there is a global service manager and a service migration service to manage Agent naming issues, when nodes connect/disconnect often, and Agent replication for reliable migrations. These are non-mobile static components that cannot be serialised for mobility. The mobile area contains a number of services that are loaded by JADE-LEAP including message transport protocols, communication protocols, and other user designed services which can be serialised for code migration to other platforms.

The next section discusses some of the new functionalities available to distributed computing applications in mobile ad-hoc networks: a new Instance Manager for soft-resetting of JADE-LEAP service components and network management, service migration, parallel threaded, and data distribution using Agents.
6.2.1 Instance Manager Thread

As described in Section 5.3.3, a new additional wrapper is proposed to run JADE-LEAP as its own manageable thread and complete the JADE-FT middleware. There are many software exceptions in the middleware and Agent applications when connecting/disconnecting to other platforms, creating Agents with the same names, missed messages, and memory overflows. On Earth, desktop PC's that operate current Agent middleware have wired connectivity, a static network topology, and larger platform resources than the resource constrained and highly mobile wireless embedded platforms targeted in this research. These errors occur all more frequently with these added restrictions which are not managed by JADE-LEAP and human/user intervention is required to correct these errors.

The Instance Manager provides additional autonomous control when exceptions occur and CDC profile management in the mobile network scenario. The one major problem with current exception handling in JADE-LEAP is that when an exception handling sequence is unsuccessful, the Java runtime environment closes. This shuts down the software loaded and stalls the processor (on a PC or on JOP). This problem is overcome by running the middleware as its own thread which can be loaded/unloaded if exception handling is not successful. So instead of typically exiting the runtime and performing a complete hardware reset, the Instance Manager first attempts to resolve the error (standard exception handling) and, if that is unsuccessful, a restart of the middleware thread under a safer profile configuration taking advantage of multiple CDC profiles (soft reset). The CDC profiles provide configurations to the JADE-LEAP Agent middleware. There is a starting configuration for safe mode operation and proactive creation of new profiles after successful network topologies are created.

An algorithm of the developed Instance Manager thread is shown as a flowchart in Figure 6-6. When started, the Instance Manager checks for the highest profile (best known network topology and connectivity state), creates a thread and loads JADE-LEAP for normal operations. In the event of an exception at loading the middleware instance or during normal operations, it is important to know if the exception a) can be handled and b) if it is expected. An example of an expected exception would be if the satellite node knows that is running out of power or drifting away from the network and a previous profile can be found to recover network services as quick as possible using the Profile N = N - 1 loop. An example of an unexpected exception would be if a failure has occurred due to SEU/SELs. In this case, all satellite nodes return to a safe mode where CDC Profile, N, = 0. The safe mode has the standard Agents services and attempts to find nearby connections from a reset network connection table.
Figure 6-6. Algorithm for the Instance Manager

Figure 6-6 shows the algorithm for managing instances of the Agent middleware JADE-LEAP. Once started, the middleware operates in normal conditions but if JADE-LEAP crashes due to an exception between two nodes, the thread is stopped and not the JVM. A loop then enables JADE-LEAP to be restarted under a different and more reliable configuration (previous described). This algorithm is achieved using new methods and interfaces shown in Figure 6-7.
Figure 6-7. Instance Manager Classes and Interface Diagram

Figure 6-7 shows the Instance Manager classes and 6 other top level classes, most notably: the JADE runtime, the JADE profile, and profile implementation interface. The Instance Manager provides a set of rules and functions that log each satellites capability (discussed in Section 6.3.2) in the network initiated at startup. The profile implementation interface allows the Instance Manager to set a number of key variables as to how to configure the runtime. These variables will determine if the satellite node is configured the main node (sink), a backup node (if the sink is removed), or a normal peer. The runtime and profile classes then load an Agent container and relevant Agents based on the chosen profile. This routine is repeated after a set time as the method: reAssign which provides another layer of abstracted control for autonomy and fault-tolerance to the distributed satellite system’s software.

6.2.2 Agent Mobility Service Overhead and Code Migration

A more complex application to test the overhead of Agent mobility is developed and studied using the method probe. Here, the software is started up with the AMS, DF (described in Section 6.2) and additional service Agents. They are then loaded for registration and management of services. Our own service is developed to mobilise class files (or functionalities) to other requesting
platforms in a wireless network. As a potentially useful satellite service a JPEG2000 compression encoder/decoder [231] is targeted for experimentation in the following sections in this chapter. In Figure 6-8, an example application where a service is distributed using the ACL scheme for Agent registration and negotiation is depicted.

Figure 6-8 considers the distributed computing scenario where a satellite requires a specific service or function but does not have it stored locally. It then multicasts out to neighbouring satellites, shown as the large arrow, in the network to find if another satellite has that service. In this example one satellite does and uses ACL messaging to negotiate the code mobility, shown as a direct link with small arrows. The service is wrapped as a mobile Agent and transferred across to the requesting satellite where it executes the service and then returns to the original satellite it came from. Instead of returning, the service can be deleted, or killed in ACL. If more than 1 of the same service is offered by multiple satellites, the first it chosen using a blocking message behaviour (discussed later in Section 6.2.3).

The method probe previously described in Section 6.1.1 is reused to display the results of this mobile Agent service, containing a JPEG2000 encoder as an example application, under the JADE-FT platform, shown in Figure 6-9. Here, the platform under test is the sending platform which wraps the mobile Agent and sends it to the requesting platform.
Figure 6-9. Memory Profile of JADE-FT with Mobile Agent Application

There are six locations where memory is increased rapidly and investigated to learn which Agent functions increase memory, as shown in Table 6-3.

Table 6-3. Classes and Methods that increase the JADE-FT RAM Requirements

<table>
<thead>
<tr>
<th>Point on Graph</th>
<th>Class/Method Name</th>
<th>JADE-FT Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Init</td>
<td>• Handles LEAP startup</td>
</tr>
<tr>
<td>2</td>
<td>Setup</td>
<td>• Agent startup (form of services)</td>
</tr>
<tr>
<td>3</td>
<td>SL.ParserTokenManager, SimpleCharStream, Ontology, ObjectSchemaImpl, CaseInsensitiveString</td>
<td>• Opens sockets for inter container communication</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Defines an Agent’s vocabulary &amp; relationship between interface elements</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Creates strings of data</td>
</tr>
<tr>
<td>4</td>
<td>AgentMobility, AgentState, Leap.LinkedList, Scheduler, Behaviour.isRunnable, Behaviour.root, Behaviour.getParent</td>
<td>• Saves state, and interface variables in an array</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Wraps &amp; schedules next behaviour</td>
</tr>
<tr>
<td>5</td>
<td>SL_CODEC, ArrayLists</td>
<td>• Codec to change between data types (strings &amp; frames)</td>
</tr>
<tr>
<td>6</td>
<td>Behaviour.init, Behaviour.restart, Behaviour.isRunnable</td>
<td>• Restarts the Agent state &amp; behaviour</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Resumes behaviour based on schedule</td>
</tr>
</tbody>
</table>
Analysis of the results in Table 6-3 has concluded that there are several memory consumptive methods used in JADE-LEAP. Points 1 and 2 in Table 6-3 are both handling the middleware startup which includes class loading and static field assignments. Point 3 handles external communication and the loading of ACL strings. Point 4 saves the Agent state and variables as well as wrapping the behaviour class and other dedicated classes (such as the compression encoder). Point 5 converts the wrapped saved Agent to frames of strings which requires using set arrays for buffering. Point 6 resumes the Agent behaviour and disregards unwanted or duplication classes. For future memory savings, buffering of the saved and wrapped Agent which uses around 450 kB could be investigated. This 450 kB also includes the existing ACL definitions into string arrays which uses approximately 200 kB to ensure language definitions match across platforms. Further memory savings of around 50 kB could be made if class duplication is avoided but this is not practical in most cases where a wireless link uses duplication to confirm successful Agent mobility in the event of lost packets/messages.

6.2.3 Parallel Threading and Agent Behaviours

Taking advantage of Java threading can be achieved using Agents for control. In this section, it is shown that parallel behaviours of newly created Agents can also be used to help parallelise tasks. Code was developed to implement Agents as Java threads for a parallel behaviour, essentially creating 'child Agents', where each child will run a distributed computing task requiring multiple connections. Figure 6-10 shows the software components used to complete a service Agent and the possible behaviours it could use.

![Figure 6-10. Service Agent Components](image)
Figure 6-10 shows how the existing JADE-LEAP behaviours are integrated as one application. The Agent structure defines the Agent’s lifecycle and behaviours are added to define a number of subtasks. The *one shot* behaviour is used to define multiple Agent ‘tasks’ as sub-behaviours to be run in parallel, the *sequential* behaviour is used to start the Agent to live one time only, run a specific task, then terminate. The *parallel* behaviour sets each Agent to run a task at the same time and can be used to schedule events. These behaviours and others such as the *cyclic*, *ticker*, and *threaded* behaviours provide sets of Agent control strategies to be used in services.

Figure 6-11 shows the console output flow for utilising these behaviours in one Agent application including JADE-LEAP startup services, the registration and addition of the service, and finally the creation of nine child behaviours, all created as threads at the same time.

```plaintext
12-Jul-2007 12:21:10 jade.core BaseType init
INFO: Service jade.core.management.AgentManagement initialized
12-Jul-2007 12:21:10 jade.core BaseType init
INFO: Service jade.core.messaging.Messaging initialized
12-Jul-2007 12:21:10 jade.core BaseType init
INFO: Service jade.core.mobility.AgentMobility initialized
12-Jul-2007 12:21:10 jade.core BaseType init
INFO: Service jade.core.event.Notification initialized
12-Jul-2007 12:21:10 jade.mtp.HTTPServer <init>
INFO: HTTP-RTP Using XMLParser com.sun.org.apache.xerces.internal.jaxp.SAXParserImpl$JAXP$AXParser
INFO: HTTP addresses:
http://localhost:7778/acc
INFO: Agent container Rain-Container@JADE-IMTP://ssclt001 is ready.

Agent Chris registering service "JJ2000" of type "ImageComp"
Agent Name: Chris started at 109
Set Behaviour Timestamp: 109
Dividing Tasks
Adding Parent Behaviour Time: 109
Adding Child Behaviours Time: 109
Compute tile 1/9. Time: 109
```

**Figure 6-11. Parallel Processing Example**

Figure 6-11 highlights the startup routines including starting services for Agent management, messaging, mobility, a HTTP server, and a container for Agents to reside in. The Agent, called ‘Chris’, has the type and ontology registered as an image compression service and starts to divide the task and create sub-tasks. The service here demonstrates the ability to create Agents from other Agents and running them in parallel in one JADE-LEAP middleware instance.
6.2.4 Data Distribution using Software Agents

To demonstrate the transfer a file between any two nodes, Agent negotiation is used to coordinate distributed data. Motivated by operating multiple distributed communication streams, Agents can provide an easy management structure. This gives another layer of abstracted control to provide autonomy to the distributed satellite system. With reference to the OSI layer system, Agents are used to handle the application layer whilst the IEEE 802.11 is used to handle lower layers.

Two IP based protocols, namely TCP and UDP, were selected to handle transport layer requests from the Agents to and from the application layer. For different distributed computing applications, it may be necessary to handle large files such as image data and small messages. TCP and UDP have some distinct differences that can be taken advantage of, including using TCP for reliable or ‘high priority’ data transfer and using UDP to multicast small messages for satellite cluster or formation management or applications where data is not time critical.

A TCP client/server application was embedded into Agents using the Java .net package designed as two services: FileSender and FileReceiver into one class called FileService. FileSender handles the initiation of file selection, IP address and port, whilst the FileReceiver accepts the data and receives guaranteed data using TCP acknowledgements. The Agents are used to handle Agent services and connections between platforms and also to negotiate communications between the two platforms using the Agent Communication Language, shown in Figure 6-12.

![Figure 6-12. Wireless File Distribution Service - Receiver](image)

Figure 6-12 shows the receiver end of the FileService where it is registered so it can be searched by the AMS and DF. As a new container is added, the FileService becomes available to other
remote platforms. A file transmission is then accepted from the new container’s FileService and data transmission occurs. Confirmation of the complete file is displayed on the console here and the FileReceiver returns to its wait state.

UDP peers and endpoints were also embedded in Agents just as in the FileService. The peer can multicast and receive to and from various nodes. The endpoints are used for 1 way communication, so a peer and an endpoint would be required on the distributed computing platform just as a client and server service.

Experiments investigating these services when transmitting small and large datasets over a wireless link were carried out to demonstrate Agent control. Here a desktop PC and a laptop (node to node) with IEEE 802.11 b/g are used to run and transmit the data under differing Agent communication schemes for the applications. A small dataset of 1,048 kB and a large dataset of 346 MB are used – both containing raw image data. Table 6-4 describes some of the experimental parameters and results obtained using Wireshark [232]. Wireshark, a network and protocol analyser, is used to capture live network data throughputs in our network using the JADE-FT system and mobility of Agents. Wireshark is typically used to troubleshoot network problems, examine security problems and learn/debug protocol implementations, which makes it ideal for this analysis.

Table 6-4. Experimental Wireless Parameters and Results of TCP and UDP FileServices

<table>
<thead>
<tr>
<th>Wireless Link Parameters:</th>
<th>IEEE 802.11(b), ad-hoc mode, signal strength = -66 dBm, noise floor = -94 dBm, connection speed = 4.7 Mbps</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Experiment</strong></td>
<td><strong>Time taken using TCP (s)</strong></td>
</tr>
<tr>
<td>Ave. Packet Size</td>
<td>999 B</td>
</tr>
<tr>
<td>Ave. Throughput</td>
<td>600 kbps</td>
</tr>
<tr>
<td>Small File Transfer (1,048 kB)</td>
<td>1.7 s</td>
</tr>
<tr>
<td>Large File Transfer (356 MB)</td>
<td>653.2 s</td>
</tr>
</tbody>
</table>

The results in Table 6-4 show that UDP achieved a 29% faster successful file transfer. The TCP and UDP data throughput results are shown in Figures 6-13 and 6-14 where the black line denotes TCP throughput, the red lines show TCP acknowledgements, and the green line shows UDP throughput. The x axis is time in seconds and the y axis is bits per second.
Figure 6-13. TCP Experiment using FileService

Figure 6-14. UDP Experiment using FileService

Figure 6-13 shows the use of TCP acknowledgements whilst Figure 6-14 shows the use of UDP instead of TCP which is faster to complete the data distribution task. It is noted that the difference between TCP and UDP is typically negligible. This can be explained by the use of TCP acknowledgement packets and larger headers which can seriously affect the throughput, unlike the UDP implementation which uses a ‘ping’ to check connectivity. It is also observed in TCP experiment, for every 400 packets/s sent, there are 200 packets/s returned as acknowledgements during the experiment. Factors which could cause this includes contention on the channel so there are lots of retries or the application was un-optimised. Unlike the TCP Service which buffers the entire file, the UDP service breaks down the file into different pieces dependent on a predefined buffer setting. The UDP buffer setting is variable and set at 4 MB initially, which will send 4 MB without any acknowledgement or checking at the transport or application layer. This buffer area could be of concern when distributing data as buffering data consumes time and memory resources, and therefore power.

Medium sharing between TCP and UDP for mobile ad-hoc networks (MANETs) and routing are discussed in [234] to find that it is often the case that TCP will back off when sharing with UDP. Foreseeable problems include a higher UDP packet loss rate and a significant TCP interruption rate. This same problem is also addressed more recently in [235] where they conclude that any
small message such as acknowledgements, non-error driven broadcasts and other periodic messaging is highly detrimental to MANETs.

6.3 Agent Computing Networking Functions

This section investigates expanded functionality of the developed middleware towards:

1. A topology reconfiguration methodology that uses hardware and software discovery, and

Each of these functions can aid in creating a fault-tolerant Agent middleware to demonstrate the technologies so far developed in relation to extremely embedded and highly mobile MANETs.

6.3.1 Topology Reconfiguration

In this Section, the SoC and Java co-processor system [236] is used towards a novel startup procedure to deal with situations that require a hard reset, soft reset and network topology change. Figure 6-15 shows an overview of the flow in which hardware and software resources are discovered so that network topology can be best reconfigured by better capable satellites.

Figure 6-15. Overview of the Topology Reconfiguration Scheme

Figure 6-15 has three main stages which are described below:
Stage 1: Startup FPGA Bus System & LEON3: Upon startup, each AMBA core sends its plug & play signals to the AHB arbiter that decodes the values and generates the correct select signals for memory assignment. The LEON3 is started where core identification and system memory can be discovered along with scheduled tasks (or other existing loads).

Stage 2: Startup JOP & JADE-FT: JOP loads the main memory microcode from Flash and then the Boot() method is invoked to start running Java programs. The memory is then measured, GC is performed to clean the cache and initialise internal data structures. JOP then statically loads class methods, loads a profile, and invokes the Main() method to start the Java application with arguments passed using the stored profiles to configure and optimise JADE-LEAP for satellite’s services in a given network topology.

Stage 3: Network Topology Refresh: To initialise, check or change the network topology, a rule engine is used that can soft-reset JADE-FT in differing configurations with differing services. An Agent can discover local data and identify services and existing interfaces or functionalities that the satellite has to offer the network.

As with other Agent ‘swarm’ or ‘clustering’ systems [237] [238] [239], the discovery of local node level resources is defined as the ‘Capability Function’ (CF) and often called a ‘role capability function’ (RCF). In this case, the satellite as a whole is described instead of each Agent to minimise computational overhead and time.

6.3.2 Capability Function Definition

The capability function describes in this section concatenates raw data together from the discoverable hardware and software sources. An Agent is registered with the intention of detecting as many variables about the system as possible including the bare hardware, the operating system, the JVM, network links and information from the previous stages. For distribution of a satellite’s given capability function, the available resources are encoded using a rule engine as shown in the Figure 6-16 into dynamic and static rules.

```java
/** Static Rules **/  
String CF1 = System.getProperty("java.specification.version");  
if (CF1.equals("1.1") { CF = CF + 10000; } // JRE Version (1.1 to 1.4)  
if (CF1.equals("1.2") { CF = CF + 20000; }  
if (CF1.equals("1.4") { CF = CF + 30000; }

/** Dynamic Rules **/  
int a2 = (int) r.freeMemory();  
int a3 = (int) r.totalMemory();  
int mem = (a3 - a2)/1000; // kilobyte return

/** Find Capability Function **/  
CF = CF + mem;
```

Figure 6-16. Rule Engine Code Snippets
Static rules need checking once but dynamic rules would need periodic measurement. The encoding scheme shown in Table 6-5 can give weightings to particular system properties based on the significance position in the final value, with an example shown in Table 6-5, so that the CF can be used as quick and simple as possible.

Table 6-5. Capability Function Encoding Example

<table>
<thead>
<tr>
<th>Bit 8 (MSB)</th>
<th>Bit 7-4</th>
<th>Bit 3-2</th>
<th>Bit 1 (LSB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Network Avail (ON ‘1’ or OFF ‘0’)</td>
<td>No. of AHB/ APB peripherals interfaces (4-5 = AHB, 6-7 = APB)</td>
<td>CPU Arch (1 = 16 bit, 2 = 32 bit, 3 = 64 bit)</td>
<td>JRE Version (4 = 1.4, 5 = 1.5, 6 = 1.6)</td>
</tr>
</tbody>
</table>

Table 6-5 parameters include the JRE version, the CPU architecture, the number of AHB and APB peripherals, and if there is any network connection but is primarily an example and can be expanded for other useful parameters. It is the final CF value that is used to determine which nodes are master or routing nodes in the JADE-FT middleware and can be used for a number of satellite management applications including routing applications, where only “useful” satellites are select when routing high priority data, or for when topology reconfiguration is required so that a “useful” and reliable satellite is selected as the sink to ground. Fixing each node to either a master or routing node could lead to suboptimal network topologies and reconfiguration optimises as per the CF. Additional important system properties that could be used in a CF are found in Appendix G (non exhaustive and technologically driven).

This scenario for topology reconfiguration is demonstrated in Sinalgo [241], a tool used for simulating mobile ad-hoc networks, where the CF application was implemented for cluster management and master node assignment of 100 satellite nodes (assigning the master node based on number of connections and relative distances). A circular orbit is implemented as a new mobility model in 3 dimensions within this frame space. It uses the Cartesian co-ordinate frame system which propagates the ‘next location’ for each node at ‘every round’ (no unit of time) based on a centre point, in our case, the Earth’s centre of mass. There is also a 1% three axis position error to simulate orbit perturbations.

Figure 6-17 and 6-18 show a typical launch scenario, as described in Section 3.3, so satellites are:

- First launched from a rocket/deployer and are initially close together (Figure 6-17 a),
- Drift apart and separate over time (in a sun-synchronous orbit scenario) due to orbital perturbations (Figure 6-17 b), and
- Finally creating ever changing network topologies requiring proactive checking for correct configuration (Figure 6-18).
Figure 6-17. Node Separation in Sinalgo a) Initially and b) After 3 Orbits

Figure 6-18. Cluster Formation using the CF Algorithm

Shown in Figure 6-18, the connections between the satellites as networked nodes are displayed in black lines to show where communication can occur and the chosen master node in red. Groups or clusters form under the CF application and the topologies can reconfigure dependent on mobility and connectivity but can be extended to include other parameters using modelling framework. The CF value for each node was not outputted on the screen because 1) it would fill the screen with undecipherable numbers and 2) dramatically increases the simulation time of this scenario.
6.4 Case Study: Distributed Image Compression

Initial simulations investigate whether substantial savings in time can be made to complete high performance computing tasks by distributing the task over several nodes. One example is a distributed image compression application, mentioned in Section 6.2.2, where a satellite can take an image and distribute the task of compression across multiple satellites. The image is first split into ‘tiles’ where the algorithm then proceeds to compile each tile sequentially before putting the completed image back together. Instead of working through the tiles sequentially, a multi-Agent system could be used to decrease time to complete by distributing a tile (or set of tiles) to other satellites in the network. Compression is a key technology used for satellite imagery and these algorithms can be very time and power consumptive depending on compression method. The previously chosen JPEG 2000 lossless compression algorithm is used to preserve as much data before being sent to ground. An image from one of the Disaster Monitoring Constellation (DMC) satellites build by Surrey Satellite Technology Ltd (SSTL) is used as the reference image or payload data, shown in Figure 6-19. The image is 15,636 (variable in along track) x 7,768 pixels which is 121 mega-pixels.

![SSTL DMC Test Image (SSTL DMC-II ©)](image)

Compression typically prepares tiles sequentially, first across then down the image. The proposed distributed image compression task aims to investigate the completion time through image
dissemination. An Agent service was written to split an image to find the service delay of this system for testing using the functions in Sections 6.2.2 – 6.2.4. The main body to tile the image is shown in Figure 6-20.

```java
BufferedImage im = ImageIO.read(new File("C:\image.bmp"));
int w = 2048; int h = 2048; // Select tile size
int xTiles = im.getWidth()/w; int yTiles = im.getHeight()/h;
int x = 0; int y = 0; int inc = 0;
for ( int a = 0; a < yTiles; a++ ) {
    for ( int b = 0; b < xTiles; b++ ) {
        BufferedImage sub = im.getSubimage(x, y, w, h);
        File outputfile = new File("ImageData" + inc + ".bmp");
        ImageIO.write(sub, "bmp", outputfile);
        inc++; x = x + w; // Increment & find new X position
    }
    x = 0; y = y + h; // Reset for new row & find new Y position
}
```

Figure 6-20. Code Snippet for sscSplitImage Service

The code in Figure 6-20 initially buffers the image and finds the number of tiles in both x and y axes, dependent on tile size. A tile is then selected and outputted for distribution to other satellite nodes. This is looped until the entire image has been tiled.

Table 6-6 shows the splitter service performance metrics, and Figure 6-21 shows the correlation between the average image data size and splitter service time to find the optimum tile size for distribution. It is important to note that there is an initial overhead of 20 seconds for the buffering/loading when splitting and reforming the image as well as a memory overhead to achieve this task.

<table>
<thead>
<tr>
<th>Image Tile Size</th>
<th>No. of Tiles</th>
<th>Average Image Data (kB)</th>
<th>Total Disseminated Image Data Set (kB)</th>
<th>Splitter Service Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>128 x 128</td>
<td>7320</td>
<td>49.206</td>
<td>360,187</td>
<td>149.047</td>
</tr>
<tr>
<td>256 x 256</td>
<td>1830</td>
<td>196.662</td>
<td>359,891</td>
<td>98.641</td>
</tr>
<tr>
<td>512 x 512</td>
<td>450</td>
<td>786.484</td>
<td>353,918</td>
<td>74.422</td>
</tr>
<tr>
<td>1024 x 1024</td>
<td>105</td>
<td>3145.781</td>
<td>330,307</td>
<td>62.734</td>
</tr>
<tr>
<td>2048 x 2048</td>
<td>21</td>
<td>12582.952</td>
<td>264,242</td>
<td>55.422</td>
</tr>
</tbody>
</table>
Figure 6-21. Image Data Size vs Splitter Service Time

The results from Figure 6-21 show that a tile size of approximately 512 x 512 is the most optimum for speed of this service whilst holding enough information for distribution. Important values taken forward from this experiment are the splitter service time of 74.422 seconds, the 786.484 kB image data size, and 20 second loading time. The optimum tile size found here also correlates in other academic literature, e.g. by G. Yu [243].

To find the complete delay of our distributed satellite system, the round trip delay (RTD) is required to find the complete time it would take to complete the task on a network with varying latencies on all OSI layers - from the hardware layer in a LEON3 processor up to the application layer. Messages in packet form must pass through all these layers, each with their own latencies, as shown in Figure 6-22.

Figure 6-22. Simulation Diagram for ISL Latencies (Basic Overview)

Some experimental variables taken from reports and results found throughout this research include:
• Worst-case hardware switching delay = 1.258 ns (Clock Report of LEON3 on XC3s-1500 FPGA)
• MAC access delay = 2.049 ms. (Discussed in [9]).
• Propagation through free space over 100 km

\[ D_{\text{Propagation}} = \frac{d}{c} = \frac{100,000}{2.99792458 \times 10^8} = 3.33 \times 10^{-4} \text{s} \] (6.1)

• Image Size = 15686 x 7768, File Size = 355,842 kB
• ISL Link Throughput = 1 Mb/s over 14 channels (totalling to 14 Mb/s)
• IEEE 802.11b (WiFi) Variables:
  o Bit Error Rate = \(10^{-5}\)
  o Packet Drop Rate = 90%
  o Usable Packet Size: 1500 of 2346 bytes
• Compression Rate for Flight Software = 2 MPixel/s for JPEG2000 Lossless Compression

Equations (6.2) to (6.6) from J. E. Underwood [244] are used find the average round trip delay (RTD) of a packet, in (6.2). This is a good indicator of quality of service, where decreasing RTD improves the quality of service by the end user.

\[ RTD = (D_{\text{Switch}} \times N_{\text{Nodes}} + D_{\text{Queue}} + D_{\text{Service}} + D_{\text{Propagation}}) \times (\text{ExtraPackets} + 1) \] (6.2)

Where \(D_{\text{Switch}}\) is the switching time through hardware at each path, \(N_{\text{Nodes}}\) is the number of nodes, \(D_{\text{Queue}}\) is the access latency through the MAC layer, \(D_{\text{Service}}\) is the service latency (time taken to load program and run a program with an OS and Middleware if required), and \(D_{\text{Propagation}}\) is the time taken for the packet to travel through free space. ExtraPackets is the probability of packet retransmission over the network multiplied by the RTD for a successful transmission, in (6.3):

\[ \text{ExtraPackets} = \left[ \frac{1}{1 - P(PktError)} - 1 \right] \] (6.3)

Where \(P(PktError)\) is the probability of packet error through the network, shown in (6.4):

\[ P(PktError) = \left(1 - \left[1 - \left(1 - \text{BER}_{\text{TOT}}\right)^{\text{PktTotal}}\right]\right) + P(DroppedPkt) \] (6.4)

Where \(\text{BER}_{\text{TOT}}\) is the total bit error rate a packet sees along the path, \(\text{PktTotal}\) is the total size of the packets (in our case 2340 B), and \(P(DroppedPkt)\) is the probability that a packet is dropped along the path. The queuing delay, \(D_{\text{Queue}}\), is found in (6.5):
Where $\lambda$ is arrival rate of the packets, $N_{Ch}$ is the number of channels, $Pkt_{user}$ is the usable packet size (in our case 1500 B), and $R_{user}$ is the service rate per packet. $\lambda$ is the congestion measured in terms of the number of dropped packet. The more congestion, the more packets are dropped to keep the network stable. This is shown in (6.6):

$$\lambda = \frac{0.99 \times N_{Ch}}{\frac{Pkt_{user}}{R_{user}}}$$  \hspace{1cm} (6.6)

These equations are not explicitly used to model specific communications protocols so Equation (6.2) is modified to increase the RTD from a single packet equation to this scenario with multiple packets by adding a time to transmit data based on the IEEE 802.11 characteristics, shown in Equation (6.7) and in Figure 6-23.

$$RTD_{New} = (D_{Split} + D_{Switch} \times N_{Nodes} \times \left( D_{Queue} + D_{Propogation} + D_{Transmit} \right) \times \frac{N_{Nodes}}{N_{Ch}} + D_{Service}) \times (ExtraPackets + 1)$$  \hspace{1cm} (6.7)

![Figure 6-23. New Round Trip Delay Flowchart](image-url)
Where $D_{\text{Split}}$ is the previous splitter service delay of 74.244 seconds and $D_{\text{Transmit}}$ is the length of time it would take each data segment to be sent to each node, shown in (6.8):

$$D_{\text{Transmit}} = \frac{\text{ISL Data Rate} \times N_{\text{Ch}}}{\frac{\text{Total Data Size}}{N_{\text{Nodes}}}}$$  \hspace{1cm} (6.8)

Where ISL Data Rate is 1 Mb/s and Total Data Size is the image data size (converted to bits). Equation (6.9) shows how $D_{\text{Service}}$ is found under this scenario to receive and compute the tiles:

$$D_{\text{Service}} = \frac{\text{Load Buffer}}{N_{\text{Nodes}}} + \frac{\left( \frac{\text{Total Pixels}}{N_{\text{Nodes}}} \right)}{\text{Compression Rate}}$$  \hspace{1cm} (6.9)

Where Total Pixels is the total number of pixels for the image (121 MPixels) and Load Buffer is the 20 seconds to buffer the complete image, taken from experimentation for the large satellite image in Figure 6-19. These parameters were programmed into a MATLAB m file and the major results are shown in Figures 6-24, 6-25 and 6-26.

Figure 6-24. Round Trip Delay: Single vs N Nodes

The round trip delay result in Figure 6-24 shows that task distribution does not reduce the time to complete this task, even over a distributed satellite network. The number of channels is a key limitation to the number of nodes the task is distributed to before the intersatellite link latencies become too great, here at 14 satellite nodes. This method also confirms the single node response if we divide the image size by the compression rate and add the buffering time (122 MPixels/2 MPixel per second = 61 seconds + 20 seconds buffering time = 81 seconds), shown in Figure 6-24.
Further investigation in transmission latency shows that retransmissions due to congestion and dropped packet rates can greatly increase the time to complete the task (shown in Figure 6-25). The most dramatic increase in performance is how fast the image is disseminated into tiles using the splitter service accounting for 65% of delay in this task. As shown in Figure 6-26, a reduction in just this delay of the distribution process can potentially speed up the application of on-board distributed image compression for useful distributed satellite systems scenarios. In conclusion, there are no time savings in distributing this particular task across multiple satellites and is costly in terms of initial data distribution time overheads, memory usage, and transmission power. It is recommended that this scenario not be implemented in space due to the above inefficiencies and that data is always compressed before transmission.

Figure 6-25. Round Trip Delay: Additional Retransmissions as Variable

Figure 6-26. Round Trip Delay: Splitter Service as Variable
6.5 Summary

The chapter explores Agent middleware with reference to the strict software requirements set in Chapter 4 and justifies the design of new and novel distributed computing software aiming at filling real-time requirements as well as enabling complex Agent software applications using a Java processor and Agent middleware and mobile services.

All known Java based Agent middleware platforms have been tested and compared using a new method probe to gain results on Java revision conformity to the Java processor, JOP and the static class loading (footprint), memory heap (RAM) usage for a standard testbench application. A critical analysis of each system has been carried out to find that both JADE-LEAP for pjava and MicroFIPA-OS have suitable characteristics towards networked embedded systems. JADE-FT, a new software configuration, contains the CDC Stack and JADE-LEAP-pjava with instance management has been chosen for further development using 800 kB RAM (max) and approximately 305 kB for ROM/FLASH footprint. This is highly competitive to modern day distributed computing solutions and an improvement of 72% on existing state-of-the-art Agent middleware solutions and 64% on previous CORBA solutions.

This new middleware configuration can be seen as a service oriented architecture for creating Agents for a range of functionalities including an Instance Manager to observe the Agent middleware instance for exception handling and profile management. Other services investigated include Agent migration for mobile code/services, interprocess communication, parallel threading, and data distribution using TCP and UDP for future applications have also been demonstrated into the new middleware configuration.

This chapter has developed new and novel applications based on topology reconfiguration using discoverable hardware and software metrics, determining a capability function for mobile ad-hoc network management. The topology reconfiguration scheme takes advantage of the AMBA2 arbiter and Java profiles to discover resources on board the satellite at 'node' and 'network' levels. The previously described Instance Manager can then be used to soft reset the JADE-LEAP-pjava middleware under differing operating conditions dependent on mobility, resources and loads. A capability function is introduced using a rule engine to describe static and dynamic rules on mobility and computing resources which are weighted by their significant bit position. This provides a fast and bit-efficient snapshot of a node's current resources and loads that can be easily multicast in a network.

A distributed image compression task case study has been simulated to show that no time savings can be made by distributing image tiles at 512 x 512 to various satellites. All delays under this scenario have been accounted for to simulate a LEO satellite cluster for taking and disseminating very large images. Foreseeable problems include the inefficiencies in loading time, distribution,
and transmission power costs and not the control and task management of such a task. It is recommended that this computing task not be implemented until technology progresses in delay tolerant communications and in servicing rates for image compression.
Chapter 7

7 Agent Computing Platform

Implementation on Picosatellite Testbed

Throughout this thesis, research into highly embedded and networkable Agent systems has been carried out towards SoC IP cores and network capabilities through Internet protocol based software and Agent middleware. This chapter aims at implementing and combining these topics together onto a prototype picosatellite design. Section 7.1 outlines the testbed design and summarises the restrictions to perform a real distributed satellite system that the Agent computing platform must adhere to. Section 7.2 goes through the test methodology to combining both the hardware and software before it goes onto the FPGA. Section 7.3 implements and measures the distributed test applications developed in this work. Section 7.4 demonstrates the applications developed working together in a network to emulate a satellite scenario. Section 7.5 summarises the chapter’s outcomes.

7.1 ‘S-Cube’ Testbed Design

The picosatellite testbed design is to be based on the CubeSat platform as described in Section 3.1.1. The CubeSat bus also comes in double (2U) and triple unit (3U) sizes to conform to a picosatellite deployment mechanism. The CubeSat Kit platform provides a standard COTS solution to develop new technologies. Research into all current CubeSat missions shows that reliability and simplicity are key requirements to ensure success, whilst having more complex systems as separate payloads. This design follows trends to ensure that our satellite can achieve multiple objectives: from successful deployment, establishing communications and turning on/off experimental payloads.

The picosatellite nodes must be computationally able to run code on the satellite to optimise the network’s ability to perform the mission (e.g. make decisions from complex algorithms to decide which satellite has the resources to communicate to ground) using intersatellite links. This will require additional hardware such as CPU, storage and RAM memory, along with added software resources such as an operating system, distributed computing environment and applications, all constrained in the CubeSat dimensions.
For a ‘technology demonstration’ mission, the payload design must include all the necessary components for the complete distributed computing platform, with communication capabilities, power systems and payloads. To ensure reliability in space, a new COTS based satellite bus architecture is proposed that considers the FPGA board, the IEEE 802.11 communications board and a camera as payloads, as shown in Figure 7-1 and as a systems block diagram in Figure 7-2.

Figure 7-1. Rendered Physical S-Cube Configuration (Front and Isometric Views)

Figure 7-2. Demonstrator Satellite Architecture

This new COTS architecture is primarily controlled by the Flight Module board (FM430 OBC) and uses the SoC design to act as a hardware and software mediator for differing payload modes in the mission. These differing modes can include soft resets or various sleep modes but also hard resets and on/off switching for varying duty cycles in orbit. This will ensure that payloads can be precisely controlled. Due to the COTS nature of the design, the SoC board is also used to interface between various buses such as I²C, SPI, PCI and Ethernet for this demonstrator.
Chapter 7. Implementation on Picosatellite Testbed

The following section looks at the data requirements of the CubeSat on-board systems and payloads. As far as CubeSat systems go, the preferred method is using \textit{I}C\textsubscript{2} to provide control signals around the CubeSat. \textit{I}C\textsubscript{2} is chosen for TTNC operations for a number of reasons, including:

1. \textbf{Flight Heritage:} It has been used on ALL operational CubeSat missions successfully.
2. \textbf{Speed:} Operates at 100 kbps (in standard mode). Other modes range from 10 kbps in low speed/power mode to 3.4 Mbps in high speed mode.
3. \textbf{Control:} The standard \textit{I}C\textsubscript{2} bus can control 112 devices and can be reduced or extended.
4. \textbf{Power:} Can operate at TTL or CMOS voltages (very low power).

The chosen configuration with possible bus connection is shown in Figure 7-3

![ESPACENET Demonstrator System](image_url)

Additional prototyping results towards a real distributed satellite system using the S-Cube can be found in Appendix G and H showing a preliminary CubeSat design that accommodates a distributed computing board payload with FPGA. The main results are:

- Preliminary Structure and Mass Budget (99.87 x 99.5 x 100 mm at 991.5 g)
- Orbital Considerations (3 x 1U picosatellites in sun-synchronous orbit)
Chapter 7. Implementation on Picosatellite Testbed

- Power System and Eclipse Times (orbital period = 98 minutes, eclipse = 35 minutes)
- Preliminary Power Budget (sunlit power = 779 mW, eclipse power = 122 mW)
- Solar Panels and Batteries (GaAs/Ge cells with 2 x batteries required)
- S-Cube CAD Drawings

This CubeSat design is currently in construction now and not fully built but is a good starting point to test this computing payload and other payloads.

7.2 Java Co-Processor SoC and Agent Middleware Integration

To integrate the Java co-processor into the SoC and Agent middleware and applications, a methodology is presented using all the hardware and software tools discussed in this research. There are three major stages in this flow:

1. **Simulation in Software:** This is where a common IDE is used (such as Eclipse) to compile and run various applications with the JADE-FT middleware under an emulated Java version. This would be the first attempt at ensuring common methods/classes are being used from JRE 1.1.8 or under.

2. **Emulation in Software:** JOP has an emulator called ‘jsim’ [220] to simulate the JOP processor hardware. It is here where problems in class loading and memory allocations first become apparent for embedded systems. It is also possible to check how much memory is being used on in the footprint also and the stack/RAM can be optimised (described further on in this section).

3. **Hardware in Loop Testing:** As described in the Section 4.5.5, the JOPizer compiles the classes to a microcode. It is this code that is then used to fill the memories in the RAM using GRMON as the footprint whilst checking that on-chip resources are acceptable. Again, knowing what goes in these RAM blocks will enable resources to be optimised further.

This sequence is illustrated graphically in Figure 7-4.
7.2.1 Simulation, Emulation, and Hardware in Loop JOP Optimisations

As the CubeSat was not completed in time, hardware simulation is confirmed using the GR-XC3S-1500 FPGA board LEON3 demonstration testbench which comes with GRLIB for that
board. It simulates the timing diagrams for two key parts of the design: the PROM and SDRAM devices. Figure 7-5 shows one of the final simulations obtained using this design and the bootloader loading plus nominal operations.

Figure 7-5. Post Place & Route Simulation with Simulated Hardware & Bootloader Load/Operation

Figure 7-5 shows some key events during the startup process which include:

1. At \( t = 0 \), the initial registers are set and the PROM starts to load. This is operates at 8-bit words and takes 4 memory accesses to get the full 32-bits.

2. At \( t = 407 \mu s \), the SDRAM is then loaded which will start the LEON3 application to load the footprint including setting of JOP’s input and output address registers to 0x41000000 and 0x42000000 respectively.

3. At \( t = 489 \mu s \), the LEON3 application sets the apbReg.jopStart signal to start the core. At this point the first few addresses are read by sc_io interface to find the start address as well as special pointer addresses. The main memory is then read using sc_mem interface and run till the end exception at 2041 \( \mu s \).

Only emulation in software and implementation in hardware are required now where this hardware/software design has led to various changes in the current methods used to compile and run Java files. Table 7-1 displays the major changeable components of the design that have occurred in this research.
Table 7-1. JOP Implementation Changes

<table>
<thead>
<tr>
<th>Location &amp; Hardware Variable</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>tools/com.jopdesign.sys.Jopa RAM_LEN = 256 (Changeable)</td>
<td>Hardware: Stack RAM Size Variable</td>
</tr>
<tr>
<td>src/com.jopdesign.sys.Const STACK_SIZE = 256 (Changeable)</td>
<td>Hardware: Stack RAM Size Variable</td>
</tr>
<tr>
<td>jop_conf100.vhd ram_width : integer : 10 (Changeable)</td>
<td>Hardware: Stack RAM Width Variable (i.e. 2^8 = 1 kB)</td>
</tr>
<tr>
<td>com.jopdesign.sys.Startup.java MAX_STACK = 100 (Changeable)</td>
<td>Bootloader: Used by &lt;classinit&gt; for loading large native (non-interpreted) class methods</td>
</tr>
<tr>
<td>com.jopdesign.tools RAM_LEN = 256 (Changeable)</td>
<td>Bootloader: Used by compiler to increase the RAM length usage</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Location &amp; Software Variables</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>jop/Makefile (All Changeable)</td>
<td>Compilation &amp; Addition of JADE-LEAP-pjava:</td>
</tr>
<tr>
<td></td>
<td>Change the location and name of targeted main application file</td>
</tr>
<tr>
<td></td>
<td>Add additional target source locations for compilation</td>
</tr>
<tr>
<td></td>
<td>Add additional .jar files for compilation</td>
</tr>
<tr>
<td></td>
<td>Apply classpaths to JOPizer tool</td>
</tr>
</tbody>
</table>

Table 7-1 shows some of the key changeable parts including JOP’s stack RAM size for an increased class loadable area in hardware and the addition of new jar files for the Agent middleware in software. As described in Section 4.5.5, applications are compiled to a .jop file. A verbose output file with debugging information is also possible for output. This is found in Figure 7-6 with an additional description of the output.

```
9061, 4660,
...
// util_dbgUdp
// 10: <init>()V

716636161, // 42 183 0 1
705758208, // 42 174 0
-1140137216, // 188 10 227 0
2763267, // 0 42 42 3
1521811458, // 90 181 0 2
-1258290767 // 181 0 1 177
```

// Length of application (in words)
// Special pointers address

Class.method
10 = Start address (32-bit word), then input variables
(e.g. [B] = boolean array), then return value of the method (e.g. 1 = integer)

A // W X Y Z
Where A = Bytecode (32-bit signed),
W = jvmAddress,
X = jvmHelpAddress
Y = mainAddress &
Z = addrRefStatic (Special Pointers)

**Figure 7-6. .jop Verbose Output**
To load various applications in memory GRMON can be used for memory allocation and editing and confirming console output achieved using the ‘write to memory’ function (mem address value).

Confirmation of the correct input codes to the RAM is shown in Figure 7-7 where the codes have been viewed in hexadecimal (i.e. 9050 Dec = 235A Hex, 4659 Dec = 1233 Hex, and 716636161 Dec = 2AB70001).

![Figure 7-7. Footprint Load and Initialisation Confirmation using GRMON](image)

When in running, key data can be written to or read from the JOP core’s internal APB registers:

- 0x80000e00: Start Address
- 0x80000e04: Output Address
- 0x80000e08: JOP Start Signal
- 0x80000e0a: Debug Signals:
  - Bits 0-2: DMA Status Signals
  - Bits 3-5: JOP Exception Status Signals

### 7.2.2 Implementation Performance Analysis

Multi-processor designs typically consist of uniprocessors rather than heterogeneous processor cores, as discussed in Section 2.3.1.1.3. This research is just the beginning of finding out how to measure the speed-up (or speed-down) of these parallel systems and how to leverage this coprocessor design and potential multi-bootloader schemes.

When run with the LEON3 performing no operations, the JOP core only needs to use the top of stack and not the next on stack as also seen in Figure 7-8 as signals ‘stack_tos’ and ‘stack_nos’. This technique is common in pipelined processors to ensure the data paths can be shifted to the next stage at every clock cycle. The existing 16-bit SRAM memory interfaces have been upgraded to 32-bit access in SDRAM at 2 µs access time. This improves both the instruction fetch cycle of bytecodes but could potentially be detrimental to the system with the slower shared memory access time.
Comparing to the LEON3, JOP is also significantly faster in this implementation. This is confirmed from the AHB Master bus request times of 525 ns for the LEON3 and 263 ns for JOP (99.6% increase). No modifications were made to the LEON3 bus interface but further optimisations may be possible. The measured DMA serve rate was 475 µs where the start and ready signals confirm a read or write to memory.

When running together, both processors operate at less than the desired memory access rate. The implemented SMP architecture in a shared memory environment creates a situation where the cores compete against each other and, in this case, the simple round robin scheme delivers access to the highest ranked core. A DMA miss or incorrect wait in the implemented design on the GR-XC3S1500 also leads to the DMA latency of 2 µs or worse, an incorrect read of incomplete data.

The performance can be measured using Equations (7.1 – 7.3) [245]:

\[
\text{UtilisationRate} = \text{ArrivalRate} \times \text{Time}_{\text{DMA}}
\]

\[
\text{ArrivalRate} = \frac{1}{\text{Time}_{\text{AHBrequestRate}}}
\]

\[
\text{Time}_{\text{Queue}} = \text{Time}_{\text{DMA Serve Rate}} \times \left( \frac{\text{UtilisationRate}}{1 - \text{UtilisationRate}} \right)
\]
Chapter 7. Implementation on Picosatellite Testbed

For LEON3:

\[ UtilisationRate = \frac{1}{525 \times 10^{-9}} \times 2 \times 10^{-6} = 3.8 \]  \hspace{1cm} (7.4)

\[ Time_{Queue} = Time_{DMA Serv} \times \left( \frac{UtilisationRate}{1 - UtilisationRate} \right) = 475 \times 10^{-9} \times \left( \frac{3.8}{1 - 3.8} \right) = 644.64 \text{ ns} \]  \hspace{1cm} (7.5)

For JOP, 2 cycle instruction fetch:

\[ UtilisationRate = \frac{1}{263 \times 10^{-9}} \times 2 \times 10^{-6} = 7.6 \]  \hspace{1cm} (7.6)

\[ Time_{Queue} = Time_{DMA Serv} \times \left( \frac{UtilisationRate}{1 - UtilisationRate} \right) = 475 \times 10^{-9} \times \left( \frac{7.6}{1 - 7.6} \right) = 546.97 \text{ ns} \]  \hspace{1cm} (7.7)

For JOP, 1 cycle instruction fetch:

\[ UtilisationRate = \frac{1}{526 \times 10^{-9}} \times 2 \times 10^{-6} = 3.8 \]  \hspace{1cm} (7.8)

\[ Time_{Queue} = Time_{DMA Serv} \times \left( \frac{UtilisationRate}{1 - UtilisationRate} \right) = 475 \times 10^{-9} \times \left( \frac{3.8}{1 - 3.8} \right) = 644.64 \text{ ns} \]  \hspace{1cm} (7.9)

As shown from Equations (7.4 – 7.9), the final speedup of utilising JOP on its own and not in parallel gives a speed improvement of 15.15%. In parallel, both the LEON3 and JOP operate at the same queue rates.

There could be potential protection using ahbmo.hprot and ahbmo.htrans signals which can define if data is a privileged fetch for buffering/cached data. This will, of course, increase the resources available and thus the power consumed. A better optimisation would include hardware or compiler based prefetching of instructions/data at any memory access.

As well as a combined srec footprint approach, a standard LEON3 C application is used to load the bootloaders into the correct memory areas and set JOPs start address, output address, and core start registers. There are some subtle differences between these methods, shown in Table 7-2.

<table>
<thead>
<tr>
<th>Bootloader</th>
<th>Size (kB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTEMS + JOP using C file Load</td>
<td>245</td>
</tr>
<tr>
<td>RTEMS + JOP combining srec files</td>
<td>549</td>
</tr>
</tbody>
</table>

These differences are due to how the C or C++ compilers can optimise for reduced ROM embedded environments whilst srec files are raw data codes which are un-optimised. Load time will also vary for larger data-set files, which will need to be investigated further.
7.3 Test Applications

To fully test the combined Java co-processor, a range of applications are needed to explore the real-time functionality and ensure the chosen Agent middleware can function in hard real-time. We follow the integration methodology to go from simulation, emulation and hardware.

7.3.1 RTJS Benchmark Applications

Due to the absence of true embedded Java benchmarks, the online community of JOP have developed a number of micro benchmarks to measure JVM performance, evaluating the number of clock cycles for single or sets of bytecodes [246] based on the Real Time Java Specification (RTJS). These include:

1. Sieve – Calculates all prime numbers in an array of size X, Y
2. Kfl – An industrial application based on distributed motor control
3. Lift – An automated lift controller in a factory
4. UDP Client/Server – A network application which loopbacks messages

All applications are self adjusting and stop when the benchmark exceeds the 1 second. The number of iterations per second is then calculated and output. Of particular interest is the UDP/IP Client application which must load a large number of assembled classes into the stack, as shown in Figure 7-9 from emulation.

![Figure 7-9. UDP/IP Client Application Class Loading](image)

The RTJS benchmark applications were run on various platforms including a PC, a mobile phone\(^3\), and the LEON3+JOP FPGA, found in Table 7-3. The results for LEON3+JOP, JRE 1.6, and the Mobile Phone are taken from experimental results (in black) for standard JOP for published work in [247] and [248] (in grey). The ‘X’ denotes where the information was unobtainable.

---

\(^3\) The mobile phone is an Orange SPV M3100 which operates with a 400 MHz Samsung CPU
Table 7-3. RTJS Benchmark Applications

<table>
<thead>
<tr>
<th>Platform</th>
<th>Speed (MHz)</th>
<th>Sieve (Iterations/s)</th>
<th>Kf1 (Iterations/s)</th>
<th>UDP/IP (Iterations/s)</th>
<th>Lift</th>
</tr>
</thead>
<tbody>
<tr>
<td>LEON3+JOP</td>
<td>40</td>
<td>1,394</td>
<td>4,810</td>
<td>2,595</td>
<td>4,721</td>
</tr>
<tr>
<td>Standard JOP</td>
<td>100</td>
<td>7,386</td>
<td>16,591</td>
<td>6,527</td>
<td>1,255</td>
</tr>
<tr>
<td>JRE 1.6</td>
<td>2 x 3000</td>
<td>771,011</td>
<td>2,631,000</td>
<td>762,600</td>
<td>1,974,000</td>
</tr>
<tr>
<td>Mobile Phone</td>
<td>400</td>
<td>22,734</td>
<td>82,926</td>
<td>36,352</td>
<td>X</td>
</tr>
</tbody>
</table>

The data shown in Table 7-3. The PC runs a JRE at version 1.6 and, as expected, is able to run large iterations of the micro benchmarks in comparison to the FPGA hardware implementations due to the high processor speed. These results are then normalised in Figure 7-10 to a standard 100 MHz platform to show the comparison of how each platform performed.

![Figure 7-10. Normalised Comparison of Micro Benchmark Experiments](image)

The normalised results shown in Figure 7-10 explore the overhead of running in the SoC design with Java co-processor and comparing them with other common Java platforms to find that the standard JOP platform outperforms the new LEON3 + JOP configuration by between 6% and 211%. This highly dependent on how much memory access there is in a given application caused by slower I/O than the SimpCon interface, direct memory access arbitration using the AHB bus, and the parallel nature of the SoC design. These two configurations are the only platforms that can guarantee real-time functionality as the other two implement dynamic classloading.

Both the new LEON3 + JOP configuration as well as the standard JOP outperform the PC desktop but the fastest is the mobile phone running the CDLC Java stack. This is explained by the fact that the base software stack is smaller. As the target functionality for both FPGA implementations targets the CDC stack, a larger configuration, more time is spent in both class loading and class
searching in comparison to the smaller CDLC stack. However, in applications requiring greater Java functionality at the JVM level, the mobile phone with the smaller MID profile will be unsuitable while the LEON3 + JOP or JOP platform operates the larger foundation profile. When comparing the new and the standard JOP platforms. The timing overhead is also dependent on the cache memory configurations used such as 1 kB in 4 blocks or as 16 kB in 64 blocks. This is investigated in [251] where larger cache sizes offer speeds of up to 28% but some applications seem to saturate at 4 kB cache and negligible performance improvement occurs with larger cache sizes.

7.3.2 JADE-FT Application

This section describes the final results of the completed Agent computing platform where the proposed JADE-FT middleware solution, described in Section 6.1.4, is loaded onto the JOP processor. The Instance Manager thread, explained in Section 6.2.1, is used to pass arguments such as container configuration, network configuration, services loaded and Agents initialised. A simple test on the PC successfully confirms that the Instance Manager thread passes correct arguments to JADE-LEAP-pjava and loads a ‘HelloWorld’ based output during emulation. The console output can be found in Figure 7-11.

Figure 7-11. Simulated JADE-LEAP-pjava Application

Now that a main() entry point is available to both JOP and JADE-FT, there were still some class files missing from JOP’s Java runtime environment to run JADE-FT and these would have to be invoked statically at startup. For this, 683 class files were identified and had to be added to the Agent runtime at a cost of 219 kB, a 42% increase in footprint size.

Once this was achieved, JOP’s emulator, jsim, is used to confirm operation on the hardware but the JADE-FT application fails due to heavy classloading. There are 3 key classloading methods in when the JVM starts up: the bootstrap classloader, the extensions classloader, and the classpath method. The bootstrap classloader loads standard JRE classes which are written in native code whilst the extension classloader loads platform specific extension classes to the base JVM [252].
The classpath method includes set jar files from set directory paths. In JOP, the extensions classloader and classpath method operate are treated identically which is what occurs in the experimentation where the additional runtime classes are added with JADE-FT in the makefile.

This process is shown in Appendix I where all the required classes for the complex JADE-FT application plus additional runtime environment classes are invoked at startup as extensions to the bootstrap classloader and cause an error. To debug this error, a hardware simulation in the VHDL simulator, ModelSim, was run and all the netlist signals were logged to show the error was due to a stack overflow exception. The standard JOP configuration implements interrupts and JVM throwables or exceptions by inserting a special bytecode to avoid modifying the pipelined processor system, typically considered to be a very complex and resource consuming process.

This ensures a deterministic handling system for interrupts and errors. The error is traced as in Figure 7-12 at 18 ms to JOP stack components.

![Figure 7-12. ModelSim Stack Error Trace](image)

There are two ways to try and overcome this error: 1) increase the JOP stack and 2) reduce the number of classes loaded further as shown in Figure 7-12. It can be seen from Figure 7-13 that the additional runtime classes added 89% more classes to invoke at startup on the JOP processor.
As illustrated in Figure 7-13, JOP’s stack size was increased to 8 kB and a further reduction of the missing runtime classes from 683 to 331 (down to a 43% increase) did not prove successful and the final design did not work. The key classes that need hardware implementation are estimated to include classes from java.io (72 classes), java.lang (71 classes), java.net (32 classes), java.text (40 classes), java.util (29 classes), and javax.microedition (c. 10 classes).

Further analysis discovered problems in the JADE-LEAP-pjava Agent middleware structure, programming styles, and data structure types. For conformity to Java revision 1.1.8 for running on the Java processor, JOP, many optimisations were made in Section 5.1.4 to reduce the Agent middleware’s code size but rewriting some of the Agent functionality would be required to reduce the number of variables, use shorter naming conventions, and use primitive data types instead of complex data types (such as array instead of java.util.vector). The existing JADE middleware was originally intended for the Internet and desktops with large memories. Therefore they were designed with functionality in mind and not embedded systems, especially real-time embedded systems. This is confirmed by testing the JADE-FT middleware for the J2EE Best Practices code review [253] found 324 severe and 758 warning programming issues as well as 12,689 recommendations for embedded Java. The J2EE Best Practices code review is part testing code for embedded systems under the TPTP tools (described in Section 6.1.1) which enables all source code to be checked to specific programming rules.

To recap the three key issues preventing the final design from working are: 1) available memory for static classloading, 2) a reduced code size, and 3) embedded programming styles, which all need further investigation. This is a significant finding and limitation of the combining current state-of-the-art Agent technologies in software with hardware JVMs and on-chip memories towards real-time Agent computing.
7.3.3 Power Results

This section looks at the power consumption increase of adding the Java processor, JOP, to the SoC design on the Gaisler Spartan-3 development board, introduced in Section 5.6.

Measuring at Jumper Point 12 (JP12) on the GR-XC3S-1500 FPGA board, the voltage and current are logged in Table 7-4 for a number of applications. For the LEON3, a genetic algorithm [200] is used to be representative of a highly complex and heavily loaded application. For the Java processor, JOP, the UDP loopback application is representative of the processor class loading and executing bytecode towards IP based network applications.

<table>
<thead>
<tr>
<th>FPGA Configuration</th>
<th>Current (A)</th>
<th>Voltage (V)</th>
<th>Power (W)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LEON3 (Genetic Algorithm)</td>
<td>0.459</td>
<td>3.3</td>
<td>1.5147</td>
</tr>
<tr>
<td>LEON3 + JOP (UDP Application)</td>
<td>0.466</td>
<td>3.3</td>
<td>1.5378</td>
</tr>
</tbody>
</table>

Table 7.4 highlights the main powers of operation between the LEON3 and the new LEON3 + JOP architectures. From these measurements, the JOP core has an overhead 23.1 mW when in operation which is an additional 15% power consumption. This shows that the Java processor can be added and used at a very minimal overhead to an existing LEON3 based SoC design.

7.4 Agent Computing Platform Emulation

To verify the functionality of the complete Agent computing platform applications without the built CubeSat platform, a combination of multiple hardware platforms is used with various applications. The middleware platform is loaded into a PC, a laptop, and a mobile phone.

This particular setup uses exactly the same middleware and software configurations on different platforms, highlighting the advantage of using Java for ‘write once, run anywhere’ development. Once the middleware is setup a topology tree can be formed and reformed dependent on the connectivity characteristics. This is setup where the PC is detected as the main node whilst the other systems are connecting container nodes, shown in Figure 7-14 by the PC, laptop, and mobile phone. As described in Section 7.3.2, the FPGA with JOP would not run JADE-LEAP-pjava. This is particularly important if the main container fails, the network and topology must be restored by utilising the previously discussed reconfiguration schemes and capability function.
7.4.1 Topology Reconfiguration

This section demonstrates topology reconfiguration using the CF and Instance Manager using the previously described platforms. Figure 7-14 shows the experimental setup of testing the topology reconfiguration where IEEE 802.11 ad-hoc mode is used to connect the platforms using IP addresses on a mobile phone, a laptop, and a PC desktop. The FPGA could not be used in these tests due to issues described in Section 7.3.2.

Figure 7-14. Experiment Setup for Topology Reconfiguration

In Figure 7-14, the CF function is used to reconfigure the master and backup nodes by returning this value from the node and local nodes to the Instance Manager thread. The new values are then assigned as follows:

- The node with the highest CF value is assigned as the main node and the sink to ground.
- The next 3 nodes are defined as backup nodes if a severe failure occurs on the main node. A backup node operates a UDP service that checks when a node drops out and reassigns if necessary.
- Other nodes are declared as standard nodes operating no additional management service.

Using this method, the current resource/ loads are taken into consideration based on any topology. Figure 7-15 demonstrates the functional testing of interprocess communications that can occur with this Agent middleware configuration for multicasting CF values to neighbouring nodes, registering mobile Agents as services for search, handling disconnections, and file distribution, all as separate threads.
Chapter 7. Implementation on Picosatellite Testbed

Profile No. 2 is the highest found profile to load with maximum services.

This section initialises management, communication, & mobility services.

This is where mobile service Agents are registered so other platforms can search for various services.

Calculated CF values are multicast to all nodes with sscComms & 'topic'=WirelessOnt'

Here, Platform 1 (laptop) connects to transfer a file. It disconnects once the task has been completed and the node is removed as it is no longer reachable.

Platform 2 (mobile phone) performs the same task as above.

Unreachable nodes are removed from the lookup table so that if it returns, there are no conflicts.

![Figure 7-15. PC Console output to demonstrate JADE-FT and Topology Reconfiguration](image)

As shown in Figure 7-15, various Agent functionalities are demonstrated and confirmed using the console outputs on the laptop and PC desktop. In this instance, an error due to naming of Agent can be overcome by proactive resets of the Agent lookup table.
A key area of interest is when an ad-hoc network consisting of mobile nodes perform topology reconfiguration based on the capability function – where a new master ‘sink’ node is assigned. The method probe was used again to find out the overhead of disconnecting and reconnecting middleware instances and performing soft resets of the middleware, shown in Figure 6-17.

Figure 6-17 shows a log of the memory performance when 1 node connects with another node. Areas of particular interest are timing and overheads with arrays and hash tables. With regards to the timing, it was found that at least 1 node is setup as the main node before reconfiguration so that other nodes can initially connect to the middleware instance, solving problems in initial ad-hoc discovery. If a node suddenly errors out, time is needed for the replicated named Agents to be removed from the main node lists before reconnection so all nodes connecting to the main node have an additional delay before connecting. From this point, any node can then use the capability function list as a preference list for connectivity before running as its own single node. I.e. if the recommended main node is not available, retry the next best. To check the memory, three middleware instances are connected using some key classes: the runtime instance, properties assignments, and profile implementations, as previously described in Section 5.2.1. These key classes contain methods which generate hash maps or arrays for holding information on the location and registered Agents at a cost of approximately 200 kB per Agent platform plus an original 600 kB for the first instance. Scalability is a key issue here and as the number of networked nodes increases by 1, the memory consumption also increases which is shown in Point 1 of Figure 6-15. Upon reconfiguration however at Point 2, the instance is destroyed and restarted under new conditions, in this case, as a backup node where messaging and control is not so
centralised. From Point 2, it is also observed that double the methods are called for one more additional networked middleware instance as the mobile phone is discovered and added.

In conclusion, this Agent middleware implementation provides the software services to perform many distributed computing tasks. However, the fact that each mobile ad-hoc network needs a central main node still exhibits a bottleneck in the system. Each middleware instance predictably operates at approximately 600 kB of RAM using the Instance Manager thread, as found in Section 5.1.3, with each connected node requiring an additional 200 kB of RAM.

7.5 Summary

This chapter introduces the design of S-Cube, a highly embedded and networked node towards a space sensor network scenario, which was not completely built. A full sub-system breakdown has shown that the nodes are computationally capable for running the new SoC with Java co-processor design at low power and low duty cycle for performing a number of applications in LEO, previously described in Chapter 3.

The methodology for developing real time Java and Agent applications is described providing three major levels of application analysis through simulation to hardware tools.

A number of real-time Java applications commonly called micro benchmarks have been implemented on several platforms including a standard JOP processor, the new LEON3 and JOP processor system, a mobile phone, and a desktop PC. When results were normalised to a set frequency, it was shown that the new system had reduced performance due to the parallel architecture and shared memory interface. The extend of the performance reduction is largely application dependent on how often it reads/writes to memory. The mobile phone implementation was the fastest due to its very small functionality but would not be able to run complex Agent systems as its lacks the basic Java 1.1.8 software functions required.

Running of highly complex Agent paradigms in hard real-time was attempted and not successful. To make the design work, considerable effort is required to implement bytecodes in JOP's native instruction set to handle more recent Java functions in code mobility, mathematics, and networking which equated to an 89% increase of the footprint. Experiments to overcome this problem included a further software reduction in the classes loaded to 43% increase and a larger stack implementation in the Java processor. The large addition to JOP's Java runtime only highlights the limit of real-time Java research to just the earliest of Java revisions.

Comparing the power, the LEON3 was running a genetic algorithm to be representative of a complex on-board processor task whilst the Java processor ran the UDP Server and Client. When
testing these two SoC configurations, there was negligible power consumption difference at 23.1 mW.

The final section demonstrates the Agent middleware platform differing hardware configurations where many Agent functions are demonstrated including message passing, data distribution, and service registration to find that a single sink creates a bottleneck in this scenario. The Instance Manager thread overhead is negligible compared to the original system in Section 5.1.3 and provides additional soft control of the middleware for added fault tolerance. Profiling of this functionality using the method probe also discovers a linear increase in network connections of 200 kB per networked node ensuring the middleware is scalable.
Chapter 8

8 Conclusions

This chapter reviews the research presented in the thesis and highlights the main novelty contributions to the state-of-the-art.

8.1 Thesis Overview

Chapter 2 presents the current state-of-the-art Agent systems for distributed computing along with a discussion on how to design and program Agent systems based on a layered model approach. It concludes that Agent systems are still very much in the research domain, despite much groundbreaking work but are currently not real-time due to their unpredictable and autonomous behaviours and their implementation in Java-based technologies. Many key functions though can be utilised to implement large scale distributed systems for control/communication applications under a uniform Java environment and using approved Federation for Intelligent and Physical Agents (FIPA) Standard middleware. To investigate real-time Java, just-in-time (JIT) compilation techniques and hardware implementations of the Java virtual machine were reviewed and considered to be taken forward in implementing a new distributed computing platform. Finally, wireless sensor network experiments using COTS components conclude that consideration is required for radiation tolerance/mitigation and power consumption in an embedded environment with complex software.

Chapter 3 has investigated picosatellites and their common trends to support future technologies in distributed satellite systems with fast development times, new found heritage, and cheap, low-mass COTS interfaces. These specific satellites are typically used as tools for technology demonstration of new components, sensors, and payloads. Distributed satellite system (DSS) terms are introduced and a review of the current and future satellite missions concludes there is only one true DSS, Iridium, and future missions aim at developing intersatellite link and on-board computing technologies. Simulation of multi-satellite deployment is shown to be a key area for distributed satellite systems. The string-of-pearl and Flower constellations were investigated to discover they can provide a maintainable network topology for distributed operations but concludes that exact connection times cannot be guaranteed due to existing oscillating perturbations between satellites. These new satellite systems are analogous to a highly mobile
wireless sensor networks as they have very minimal computing resources to sense and communicate in a given environment. Terrestrial distributed computing techniques such as the TCP/IP has been used in space along with custom space communication protocols (delay tolerant networking). Agents have so far been used in NASA’s RAX mission for simple controlling and proposed in TechSat-21 but never implemented due to complexities. Now technology has progress, Agents can potentially be used to provide code mobility and distributed control.

Chapter 4 brings the previous three chapters together discussing the technological drivers in distributed computing, satellite design, and distributed satellite systems. These drivers were then used for requirements at ‘node’ and ‘network’ levels with emphasis on low ROM/RAM resource consumption and increased functionality for future missions. A new distributed computing platform is proposed to meeting these requirements using an existing ESA SoC design using the LEON3 for ‘node’ functionality. JOP is selected as a Java co-processor for real-time Java functions and provides a Java runtime environment for Agent middleware and network applications. This completes the ‘Agent Computing Platform’.

Chapter 5 overviews the integration of JOP as an IP-based Java co-processor with the AMBA AHB bus. Operating in parallel with the LEON3, it provides a plug & play, real-time Java 1.1.8 runtime environment and uses simple register modification/polling for interprocess communication. Alternatives to this implementation are presented using a second dedicated AHB bus and via the APB bus. Compared to other similar solutions, this new SoC design can reduce the footprint by 37%. Interfacing, addressing, and exception handling are solved for bootloading of the LEON3 and JOP processors. Timing and resource constraints are met on a Spartan-3 FPGA development board at an estimated 291 mW and 24% LC increase. The design is resilient to the extreme LEO temperatures but radiation mitigation using triple modular redundancy (TMR) for flight readiness requires a larger device.

Chapter 6 introduces the first critical analysis on all FIPA compliant Agent platforms. A new software probe has been developed to find the class, method, and memory consumption to accurately find the memory consumption of each platform and where improvements can be made. The final software configuration has been designed as a service oriented architecture from the CDC stack and JADE-LEAP with a footprint of 305 kB which is 72% smaller than existing Agent solutions and 64% smaller than a comparable CORBA implementation. A range of functionalities are also experimented with such as exception handling and profile management using a new Instance Manager for soft resets by destroying erroring Agent middleware instances and restarting under a safer configuration – completing the JADE-FT configuration. Agent migration or code mobility, parallel threading, and data distribution are also demonstrated. A novel topology reconfiguration scheme based on discoverable hardware and software signals to provide multiple operational modes using a capability function. The capability function described here is
introduced to log and distribute mobility and computing resource readings to other satellites. A rule engine is then used to set master, backup, and peer nodes. The concept of distributed image compression across many satellite nodes is investigated as a case study to conclude that there are no time savings made when distributing tiles for processing and there are significant overheads in memory, time, and transmission power when serving this application. Unless service rates and communications technologies are improved, it is not recommended to implement this task.

Chapter 7 describes the design of a picosatellite, S-Cube, to provide mass budget, volume, and power limitations as a suitable technology demonstrator for the new Agent computing platform through orbit, solar panel, and battery considerations. Even with COTS components, the design was not completed in time and measurements were made on the Spartan-3 FPGA board. A methodology for integrating and testing the new SoC design with the Agent middleware is presented to speed up Java application development in software simulation, emulation, and hardware. Real-time Java applications on JOP have been confirmed in hardware simulation to find the interface is almost twice the speed of the LEON3 interface to the AHB bus. Running a test of microbenchmarks on the FPGA and other platforms are compared with the each other and the literature to verify the Java co-processor’s operation. It concludes that adding the Java co-processor reduces performance due to shared memory. Confirmation of the Agent middleware running on the SoC design could not be achieved. Limitations of the current Java processors include heavy static classloading due to incomplete native instructions and small stack implementations. It was also observed that the structure of the Agent middleware components was designed for functionality and not for embedded systems.

Final remarks to this research are that Agents are still looking for that killer application and their autonomous behaviour is more of a hinderance to designers, especially in real-time systems. The richness of some Agent functionalities however can provide key middleware and software services which are comparable to existing solutions but complex distributed applications and systems discussed in this thesis need further development.

### 8.2 Contributions to the State of the Art

This work has contributed in the following areas:

- The design of a novel Agent computing platform for on-board real-time Java functionality which combines the Java optimised processor, JOP, as a plug & play intellectual property core in the flight proven LEON3 system-on-a-chip design library. Demonstrated to operate in a Spartan-3 FPGA, this core could be used in any AMBA-based embedded design.
Chapter 8. Conclusions

- A new Agent middleware configuration with instance management functionality for software-exception tolerance and network management using message passing schemes which is unavailable in current state-of-the-art Agent middleware. Code migration, parallel behaviours, and data distribution services are included in the small 305 kB footprint which is smaller than comparable software designs for future applications in mobile ad-hoc sensor networks and satellite sensor networks. Under test, it consumes a predictable 600 kB RAM with 200 kB for each networked Agent middleware instance.

- The concept and systems design of a picosatellite testbed as a distributed networked node to support the new Agent computing platform for fault-tolerant networking applications, which enables a hard real-time Java environment when combined with the JOP processor.

Other contributions include:

- A Low Earth orbit extension and high precision simulation of the Flower constellation scenario using multiple picosatellites to show that, despite their predictability, ad-hoc networking is still required to deal with periodic disconnections.

- The first major collection and review of modern picosatellites launched since 2003 highlighting trends in success, on-board computing, and mission payloads.

- A comparison and analysis of all FIPA compliant Agent middleware platforms so far unseen in any literature, exploring footprints, start-up times, and memory consumption.

8.3 Publications

This section highlights the published journal and peer reviewed conference papers as contributions from this research.

Journal Papers:


Refereed Conference Papers:


8.4 Future Work

This section presents a preliminary roadmap for the future directions this research could generate in networked embedded systems, specifically in scientific drivers and technology drivers as shown in Figure 8-1. The need for new scientific computing architectures will be based on finding that killer application for Agent technologies in the fields shown. Technology drivers for modern day electronics will continue with Moore’s law and market trends and distributed computing systems, such as the (almost) ubiquitous mobile phone industry, could potentially create new
commercial and terrestrial applications in real-time Agent computing. Industrial validation using a number of platforms is foreseen to help mature this field over the next 20 years.

**Scientific Drivers**
- New hardware based ‘virtual machine’ architectures
- New software paradigm application designs
- Automatic hardware/software change detection & repair
- Internet/Semantic web based Agent service provision

**Technology Drivers**
- IP Cores for interfaces & application specific designs
- Reduced power and cost
- Increased reliability, on-chip memory, & logic area
- Partial Run-time configurability
- Automatic & mature design & test tools
- AI/‘Chaotic’ software design & test tools

**Industrial Validation**
- Wireless sensor network/mobile ad-hoc network implementations & results [1-5 years]
- Mobile phone or games console applications [5-10 years]
- Autonomous Robotics (domestic/military) [5-10 years]
- Ubiquitous computing [10-20 years]

Figure 8-1. Roadmap to Validating Real-Time Agent Computing

### 8.4.1 Hardware-based Java Virtual Machine Assembly Codes

Areas which require immediate work includes the implementation of ‘standard’ assembly codes which would bring the JRE to a minimum of version 1.4 whereby more functionality such as Agent mobility, various mathematic functions, and new networking functions (including security) could be utilised in newer Agent and MANET applications. Including the classes mentioned in Section 7.3.2, further dedicated Agent specific codes could also be implemented at a hardware level (c. 92 classes).


8.4.2 Classloading and Scheduling

Class Loading is a major issue in hardware based Java virtual machines. This work has, so far, provided the basis for developing Java applications in various runtime configurations to great effect but class loading is so far completely static and un-optimised for embedded devices. Dynamic class loading is still under development that could further reduce the memory requirements in very resource constrained networked embedded systems. Integrated class loaders and schedulers could be a major area for improvement under scenarios dependent on timing constraints. Also, the Agent middleware investigated were also not highly optimised for embedded networked devices.

The use of Agent technologies has also been investigated but many application scenarios need further work. Re-implementing some common tasks, such as autonomous image scheduling, typical satellites have to carry out in Java could help developers debug faster, implement in hardware faster along with further benefits. The results used for service and Agent code mobility can be used in the future to investigate different methodologies of this memory intensive code mobility operation. Agent based parallel threading methods could be especially useful in evolving or evolutionary software systems where software can create and destroy software components as they are needed. Although this is counter intuitive to the ‘mission-critical’ application domain – fast parallel systems are still a major research area that if run in real time could be able to form new methods of task management.

8.4.3 MANET Algorithm Simulations

In simulation, Sinalgo models developed have so far only included a circular orbit. Future models could theoretically include antenna beams, different interference models, differing mobility models and other network characteristics. Investigating the relationships between mobile nodes with different network models could also provide new findings and develop new models.

The capability function could also benefit from other discoverable values based on satellite telemetry and other SoC signals (at node level) but also network level values. Research into MANET based routing algorithms could also be included.

8.4.4 Radiation Effects

In this research, the space environment has been discussed but has omitted radiation effects on on-board computers and rectification methods. In most cases mitigation is never fully achievable and it is important to understand and be able to predict the radiation environment dependent on the orbit. Space radiation can never be completely mitigated on any satellite system but hardware and design can be employed to protect the satellite by fault-tolerant design and manufacture.
Software techniques such as ECAD and TMR could be used to check for SEE issues. It is typical for satellites to orientate the internal components to protect vital or vulnerable components and shielding is used where possible. So there are many methods for protecting the satellite. Longer missions will require addressing TID and displacement damage effects at the physical component level.
References


[18] P. Braun and W. Rossak, “Mobile Agents – Basic Concepts, Mobility Models & The Tracy Toolkit”, Book, Published by Elsevier Inc. (USA), 2005


[64] P. Braun and W. Rossak, “Mobile Agents – Basic Concepts, Mobility Models & The Tracy Toolkit”, Section 2.1. A First Look at Mobile Agents, pp. 7-17, published by Elsevier Inc. (USA), 2005


References


References


[107] “ESA Wireless Study”, Technical Note No 1, 8th December 2006, Surrey Space Centre, University of Surrey, Guildford, UK


[120] “ESA Wireless Study”, Technical Note No 2 – Mote Technology and ‘Smart Dust’, 30th July 2007, Surrey Space Centre, University of Surrey, Guildford, UK

[121] “ESA Wireless Study”, Technical Note No (Test Report in case of [121]), 30th July 2007, Surrey Space Centre, University of Surrey, Guildford, UK


179


[141] University of Tokyo, “University of Tokyo CubeSat”, Website [Online]. Available: www.space.t.u-tokyo.ac.jp/cubesat/index-e.html (last accessed: 17.06.2009)


References

[155] M. Maleski and A. E. Zimbler, “Internetworking through Milstar” in Proc. for Military Communications Conference (MILCOM ’95), San Diego, CA USA, pp. 474-478


References


References


## Appendix A. WSN Mote Kits Comparison

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Millenium</td>
<td>2 x HCS08 (2.1V/40MHz)</td>
<td>Missing</td>
<td>Industrial</td>
<td>916M / 2.4G</td>
<td>35m / 42m / 6u</td>
</tr>
<tr>
<td>Jennic</td>
<td>JN5121 (2.2V/16MHz)</td>
<td>64k / 96k</td>
<td>-40 to 85</td>
<td>2.4G</td>
<td>45m / 50m / 14u</td>
</tr>
<tr>
<td>DUST Networks</td>
<td>M2030 (3V)</td>
<td>Missing</td>
<td>-40 to 85</td>
<td>900M / 2.4G</td>
<td>20m / 22m / 10u</td>
</tr>
<tr>
<td>MicroDAQ</td>
<td>? (3V)</td>
<td>32k / 64k</td>
<td>0 to 50</td>
<td>900M / 2.4G</td>
<td>Missing</td>
</tr>
<tr>
<td>Ember</td>
<td>EM260 (3.3V/24MHz)</td>
<td>Missing</td>
<td>-40 to 85</td>
<td>2.4G</td>
<td>20.7m / 19.7m / 0.5u</td>
</tr>
<tr>
<td>Zensys</td>
<td>ZW0201 SoC (3.3V/16MHz)</td>
<td>2k / 32k</td>
<td>-30 to 85</td>
<td>908M</td>
<td>23m / 21m / 2.5u</td>
</tr>
<tr>
<td>Crossbow (Telos)</td>
<td>MSP430 (3.3V/8MHz)</td>
<td>48k / 1M</td>
<td>-40 to 123.8</td>
<td>916M / 2.4G</td>
<td>23m / 21u&gt;1u</td>
</tr>
<tr>
<td>TinyNode</td>
<td>MSP430 (3.3V/8MHz)</td>
<td>48k / 1M</td>
<td>-40 to 85</td>
<td>870M</td>
<td>33m / 14m / 1u</td>
</tr>
<tr>
<td>Sensinode</td>
<td>MSP430 (3.3V/8MHz)</td>
<td>10k / 4M</td>
<td>Missing</td>
<td>2.4G</td>
<td>Missing</td>
</tr>
<tr>
<td>Intel Mote 2</td>
<td>PXA27x (0.85V/13MHz)</td>
<td>256k / 32M</td>
<td>0 to 70</td>
<td>2.4G</td>
<td>17.4m / 18.8m</td>
</tr>
<tr>
<td>BT Nodes</td>
<td>Attmega12L (3.3V/8MHz)</td>
<td>128k/244k + 4k EEPROM</td>
<td>Missing</td>
<td>433M / 916M</td>
<td>42m / 29m</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Millenium</td>
<td>115 kbps</td>
<td>30m</td>
<td>802.15.4</td>
<td>MeshScape, API Lib</td>
<td>$4500 for 9P (EU)</td>
</tr>
<tr>
<td>Jennic</td>
<td>Missing</td>
<td>400m (&gt;4km)</td>
<td>802.15.4, ZigBee</td>
<td>GNU Toolchain, MCU Lib</td>
<td>$999 - 1299 for 5P (305 UK)</td>
</tr>
<tr>
<td>DUST Networks</td>
<td>250 kbps</td>
<td>200m (&gt;400m)</td>
<td>802.15.4</td>
<td>Network GUI, API Lib</td>
<td>Missing (For 13P)</td>
</tr>
<tr>
<td>MicroDAQ</td>
<td>Missing</td>
<td>75m (&gt;750m)</td>
<td>802.15.4, ZigBee</td>
<td>Network GUI for Control</td>
<td>$2999 for 9P (US)</td>
</tr>
<tr>
<td>Ember</td>
<td>250 kbps</td>
<td>75m</td>
<td>802.15.4</td>
<td>InSight GUI</td>
<td>$2500 for 10P (UK)</td>
</tr>
<tr>
<td>Zensys</td>
<td>Missing</td>
<td>60m</td>
<td>None</td>
<td>ZWave Protocol, API Lib</td>
<td>Missing (Asia)</td>
</tr>
<tr>
<td>Crossbow (Telos)</td>
<td>250 kbps</td>
<td>100m</td>
<td>802.15.4</td>
<td>TinyOS + Assoc Progs</td>
<td>£87 each / $780 for 10P (US)</td>
</tr>
<tr>
<td>TinyNode</td>
<td>250 kbps</td>
<td>200m (&gt;2km)</td>
<td>None</td>
<td>TinyOS + Assoc Progs</td>
<td>£609 for 5P (EU)</td>
</tr>
<tr>
<td>Sensinode</td>
<td>250 kbps</td>
<td>100m</td>
<td>802.15.4, ZigBee</td>
<td>Nanostack Protocol + API</td>
<td>£2895 for 17P (EU)</td>
</tr>
<tr>
<td>Intel Mote 2</td>
<td>250 kbps</td>
<td>&quot;Low&quot;</td>
<td>802.15.4, 802.11, Bluetooth</td>
<td>TinyOS + Assoc Progs</td>
<td>Research only</td>
</tr>
</tbody>
</table>

186
Appendix B. Embedded Operating Systems

B-1. Operating System Terminology

- **Virtual Machine** – This is a set of abstractions used to access resources, such as memory or registers, between the hardware and the designer to simplify and support development. It provides a pool of system wide information for reading/writing to by any process in the system. The Java Virtual Machine (or JVM) provides utilities, known as applets, for sharing applications over multiple machines in machine code or user applications.

- **Resource Management/Scheduler** – The scheduling and control of multiple users or peripherals sharing the same resources (processor, memory, storage, network ports, etc) is resource management. This shared access allows a faster performance of complex tasks and instructions and quicker time to platform or end user. An example is using device drivers to manage I/O where the software will only allow privileged users to access the ports. Decisions need to be made when a process becomes blocked, time slice for running a process expires or a process is pre-empted. There are 3 types of scheduling methods: FIFO, Round-robin and adaptive.

- **Interrupt Nesting** – Processes and applications can give out an unscheduled and unpredictable event, creating an interrupt and starting an interrupt service routine (ISR). In complex and fast systems, multiple interrupts can also occur in very short periods of time. The OS must have the ability to manage and prioritise these ISRs in a predefined block of memory. *Interrupt nesting* is where there are interrupts within ISRs and OS’s must be able to operate in real-time systems.

- **File System** – The OS here defined a hierarchical file system of virtual memory/storage to aid designers. Often blocks of memory (or pages) are reserved or protected for operating system, ISR memory, user programs and data. In the case where there are multiple users, for instance in a distributed system, the user areas are protected from overlapping each other. Switching between users means that caches are constantly flushed and refilled with relevant user data (adding to power costs).

- **Memory Management Unit (MMU)** – The MMU is responsible for handling memory accesses requested by the CPU. *Virtual Memory Management* and *Memory Protection* are but a few of the functions of the MMU.

B-2. Open Source Operating Systems

RTEMS [4]: An acronym for real-time (operating system) for embedded multiprocessor systems, RTEMS supports large numbers of API and interface standards. It is an open-source operating system under the GNU project (unofficially) and aims to create new ports, architecture and CPU models is linked with developing more support in debugging, code libraries and infrastructure improvements. Basic kernel features include multitasking capabilities, homo and heterogeneous multiprocessor systems from an event-driven, priority based pre-emptive schedule. It can network many client services, servers and middleware interfaces together and comes with many ‘thread aware’ debugging tools including GNU debugger over serial, parallel or ethernet ports to many target environments (from ARM, TI C3X/4X, Hitachi H8, HP i386, PowerPC, Sparc V+ and MIPS 32/64-bit CPU models).

Typical small applications are often 64-128 kB on any target environment with BSP, device drivers and C library. Larger apps with middleware, networking and message buffers (especially

TCP/IP buffering) could use over 512 kB RAM [5]. It has a massive online community from the open development environment of this particular OS with many webpages of explanatory notes, hints and tips for using and building in RTEMS.

TinyOS: This free software was developed after the creation of the Mica motes to cater for their unique hardware characteristics. It is a component based, highly configurable OS with a very small footprint [6]. It interconnects components with a FIFO scheduler, where components communicate through commands and events. Commands communicate downwards to low level (simple) components and events propagate upwards and are dealt with at a higher level (such as in interrupt from hardware). A component consists of a set of tasks, event handlers and a memory allocation called a frame (this stores the state of the component/ events). A minor disadvantage is that TinyOS is not full real time.

It is programmed in a language, called nesC, which ‘wires’ components and modules together but also creates many new file types (comp and desc files) which can make work tedious and error-prone. Generic components are available with TinyOS allowing great flexible design methods, but on the downside, customising/ renaming of components involves the modification of many separate files to keep the interface/wiring synchronised. There are visualising tools available include GRATIS [7] and a TinyOS Plug-In for Eclipse which both displays the interconnections and wiring hierarchically for easy design and debugging.

uCLinux (or Embedded Linux): This monolithic operating system has become popular on embedded systems because of the modular nature of linux. Thus the user can remove utility programs and other system services that are not required in an embedded environment to reduce the footprint and improve efficiency. One characteristic of uCLinux is that any procedure can call any other procedure, and therefore lacks modularity.

Snapgear [8] is a uCLinux derivative specialised to embedded systems. It differs from other OS solutions by supporting memory-management-unit less MPUs (which would not work in standard uCLinux). This freeware come with standard libraries/ toolsets, and supports the latest MPUs such as the Hitachi SuperH family.

eCos [9] is a real time OS which as been used widely in many embedded system designs. It includes an eCos kernel, a packet router library and other custom component based application development. E.g. there are 9 adapted modules for space flown on DTU-Sat [10] (housekeeping collector, telemetry manager, space radio protocols, attitude, power and camera modules). Redhat Linux also released eCos making it open source, royalty free and license free.

FreeRTOS [11] is an extensively ported mini real time (pre-emptive and co-operative) kernel for embedded systems. Currently on version 4.0, it is constantly being updated with ports to the latest hardware architectures (more recently including the MSP430 MCU and ARM9 processors). With royalty free detailed C code, the website contains sample/ demo solutions and educational pages on fundamentals and complications of OS use. One feature includes FreeRTOS using ‘co-routines’ instead of tasks so that they share stack memory reducing the RAM required for operation. In larger projects, FreeRTOS would be too simple and lack expandability for increased complexity and other OS solutions would be considered.

B-2-2. Commercial Operating Systems

[8] Snapgear Website, [http://www.snapgear.org/snapgear/about.html](http://www.snapgear.org/snapgear/about.html)
QNX Neutrino v6.2 RTOS [12]: Used in medical instruments and emergency call centres, this RTOS is industrially recognised for its reliability and fault-tolerance. It uses a microkernel operating system where drivers, applications, protocols and file systems runs outside the kernel so if any operation fails, components can be automatically restarted without affecting the kernel (or other components). Aimed at systems beyond 4G technology, it maximises the memory capabilities of all major chipsets (ARM, PowerPC, x86, etc) and can ‘self-heal’ or recover from faults to increase quality of service (QoS). It is fully scalable to allow simplification to designs in cluster groups to integrate other CPUs or port other OS’s.

VXWorks (Wind River) [13]: Now on Version 6, VX Works is the worlds most widely used commercial real-time OS used in the aerospace industry. It also works closely with NASA JPL for their Mars Rover OBC systems and ESA’s PROBA satellite. New features in V6 include enhanced memory protection, error management, improved OS scalability and supports latest networking and security protocols. A unique process based mode (as well as a standard kernel) is available to simplify application development. A processor abstraction layer (PAL) provides OS support to other similar architecture families. The scalability can be reduced to a minimal kernel profile at 36 kB but with many features removed (dynamic memory allocation, watchdogs). Each design can be customised by enabling or disabling individual components to meet device requirements.

Windows CE [14] (sometimes abbreviated WinCE) is a variation of Microsoft's Windows operating system for embedded systems providing a component-based, embedded, real-time operating system. Windows CE has a distinctly different kernel, rather than a "trimmed down" version of desktop Windows. It is supported on Intel x86 and lookalikes, MIPS, ARM, and Hitachi SuperH processors. Obviously, being part of Microsoft, the development tools [15] and user interfaces are very friendly with an Application Builder (in .NET or embedded C++) and a Platform Builder (to customise run time images). WinCE (and all Microsoft products) also benefit from constant patches/ shared development problems as it has a large market share.

Appendices

Appendix C. Flower Constellation Methodology

1. Repeating Ground Track

To begin, an orbit is defined so that the ground track repeats after one complete orbit of Earth. This orbit will have a compatible orbital period or nodal period, $T_{\Omega}$, to match the nodal period of Greenwich, $T_{OG}$. This is extended to find the period of repetition, $T_r$, where a satellite will repeat $N_p$ revolutions of $N_d$ days, described as:

$$T_r = N_p T_{\Omega} = N_d T_{OG}$$  \hspace{1cm} (C-1)

In this methodology, steps are followed to find the nodal period including perturbations, solving for classical orbital parameters ($a, e, T$), and find the satellite phasing based on the mean anomaly and right ascension of the ascending node.

2. Finding the Nodal Period

This is defined by Carter\(^{16}\) as:

$$T_{OG} = \frac{2\pi}{\omega_0 - \Omega}$$  \hspace{1cm} (C-2)

Where $\omega_0 = 7.29211585530 \times 10^{-5}$ rad/sec

To include perturbations, the nodal period of the satellite is a function of its anomalistic period, $T$, as follows:

$$T_{\Omega} = T \left(1 + \frac{\dot{\omega}_0 + \dot{\omega}}{n}\right)^{-1}$$  \hspace{1cm} (C-3)

Where the rate of change in the mean anomaly, $\dot{\omega}_0$, and the rate of change in the argument of perigee, $\dot{\omega}$, and the satellite’s mean motion, $\Omega$. This is based on geo-potential perturbation theory considering second order zone effects\(^{17}\).

$$\dot{M}_0 = -\xi \cdot n \sqrt{1 - e^2} \left(3 \sin^2 \iota - 2\right)$$  \hspace{1cm} (C-4)

\(^{16}\) D. Carter, “When is the Groundtrack drift rate zero?”, CSDL Memorandum EDS-91-020, 1991, Cambridge, MA, USA at Charles Stark Draper Laboratory.

\[ \dot{\omega} = \xi \cdot n \left( 4 - 5 \sin^2 i \right) \]  
\[ \dot{\Omega} = -2 \xi \cdot n \cos i \]  
\[ \text{where } \xi = \frac{3 R_\oplus^2 J_2}{4 \mu^2} \]  

(C-5)  
(C-6)  
(C-7)

The mean radius of the Earth, \( R_\oplus = 6,378.1363 \) km, \( J_2 = 1.0826269 \times 10^3 \), \( \mu \) is the orbital parameter, and \( i \) is the orbit inclination.

Like us, Vallado considers circular orbits (i.e. \( e \approx 0 \)) but if highly elliptical orbits are considered, the eccentricity terms must be kept. Using Equations (D-3) to (D-7), we find:

\[ T_\Omega = T \left[ 1 + \xi \left[ 4 + 2 \sqrt{1 - e^2} - \left( 5 + 3 \sqrt{1 - e^2} \right) \sin^2 i \right] \right]^{-1} \]  

(C-8)

3. Solving for \( a, e, \) and \( T \)

If we assume a orbit with a repeating ground track of 1 sidereal day, 14 revolutions/day, an inclination of 116.6°, and a height of perigee, \( h_p \), of 686 km. The semi major axis, \( a \), is resolved as a function of eccentricity, \( e \), as shown below:

\[ e = 1 - \frac{R_\oplus + h_p}{a} \]  
\[ p = a(1 - e^2) = 2(R_\oplus + h_p) - \frac{(R_\oplus + h_p)^2}{a} \]  

(C-9)  
(C-10)

The anomalistic period, \( T \), and mean motion can now be solved by:

\[ T = \frac{2\pi}{n} = 2\pi \sqrt{\frac{a^3}{\mu}} \]  

(C-11)

where \( \mu = 398,600.4415 \text{ km}^3/\text{sec}^2 \).

4. Satellite Phasing

The Flower Constellation phasing is critical to achieve the desired effect. There is a direct relationship between the right ascension of the ascending node (RAAN) and the mean anomaly at the initial time. In the ECF frame, the first satellite’s initial orbit position is characterised by the mean anomaly, \( M_i \), and the RAAN, \( \Omega_i \) at time = 0.

Other satellites can be phased based from this first satellites initial position at \( (\Omega_i, M_i) = (0, 0) \). Additional satellites in the constellation will intersect the initial position after a time interval:

\[ \Delta t = \frac{\Omega_i - \Omega_j}{\omega_\oplus + \Omega} \]  

(C-12)
Appendices

Where \( \omega_b \) is the Earth spin rate and \( \dot{\Omega} \) is the nodal procession rate of change due to perturbations such as \( J_2 \). So after a \( \Delta t \) time the mean anomaly has increased in value by:

\[
\Delta M = (n + \dot{M}_0) \Delta t
\]  

(C-13)

For a symmetrical Flower Constellation scheme, this set is then generalised so that the RAAN and mean anomalies can be found using the following equations:

\[
\Omega_{k+1} = \Omega_k - 2\pi \frac{F_n}{F_d}
\]  

(C-14)

\[
M_{k+1} = M_k + 2\pi \frac{F_n N_p + F_d F_h}{F_d N_d}
\]  

(C-15)

where \( N_p \) = no. of petals, \( N_d \) = no. of days to repeat the ground track, \( N_s \) = no. of satellites, \( F_n \) = phase numerator, \( F_d \) = phase denominator, and \( F_h \) = phase step.

The phase numbers are chosen arbitrarily but minimisation of the average coverage gap duration is done by setting \( F_n F_d = N_s \) to evenly space the satellites.

It is important to note that in this example, the mean anomaly = true anomaly if a circular orbit is assumed. If an asymmetric scheme is preferred, it is best to set \( \Omega \) as a variable when designing with intersatellite link connectivity in mind by checking the relative range at the equator and starting with the highest value first.
Appendix D. VHDL Code for SoC Design

1. Top Level Component Instantiation (leon3_jop.vhd)

```vhdl
jop0 : jopIPcore
generic map (pindex => 14, paddr => 14, hindex => 4)
port map (rstn, clk_m, apbi, apbo(14), ahbmi, ahbmo(4));
```

2. JOP Wrapper code (jopIPcore.vhd)

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;

library jop;
use jop.jop_types.all;
use jop.sc_pack.all;
use jop.jop_config.all;

library grlib;
use grlib.amba.all;
use grlib.stdlib.all;
use grlib.devices.all;
library gaisler;
use gaisler.misc.all;

library work;
use work.all;

entity jopIPcore is

generic (
    hindex : integer := 0;
    hirq : integer := 0;
    pindex : integer := 0;
    pirq : integer := 0;
    paddr : integer := 0;
    pmask : integer := 16#FFF#
    jpc_width : integer := 11;

```
block_bits : integer := 4 ;

end jopIPcore;

architecture rtl of jopIPcore is

begin

-- 1) COMPONENT DEFINITIONS

component jopcpu
  generic (
    jpc_width : integer := 11; -- addr bits of java bytecode pc = cache size
    block_bits : integer := 4 -- 2^block_bits is number of cache blocks
  );
  port ( clk, reset : in std_logic;
    sc_mem_out : out sc_mem_out_type;
    sc_mem_in : in sc_in_type;
    bc_len_out : out unsigned(jpc_width-3 downto 0);
    sc_io_out : out sc_io_out_type;
    sc_io_in : in sc_in_type;
    irq_in : in irq_bcf_type;
    irq_out : out irq_ack_type;
    exc_req : out exception_type );
end component;

component fault
  port ( clock : in std_logic;
    reset : in std_logic;
    enable : in std_logic;
    spov : in std_logic;
    ab : in std_logic;
    np : in std_logic;
    reset_ft : out std_logic );
end component;

signal enable, reset_ft : std_logic;

-- 2) STATE MACHINE REGISTERS

-- APB Configuration
constant REVISION : amba_version_type := 4;
constant pconfig : apb_config_type := (
  0 => ahb_device_reg (VENDOR_JOP, JOP_CORE, 0, REVISION, pirq),
  1 => apb_iobar(paddr, pmask));

type apb_type is record
  startJOP : std_logic;
  startAddr : std_logic_vector(31 downto 0); -- JOP PROM Start Address
  outAddr : std_logic_vector(31 downto 0); -- JOP PROM Incr Address
end record;

signal apbReg, apbIn : apb_type;

-- AHB Configuration
signal dmai : ahb_dma_in_type;
signal dmao : ahb_dma_out_type;

type dcom_state_type is (idle, load, read, write);
type ahb_type is record
  addr : std_logic_vector(31 downto 0); -- Address Register
  Raddr : std_logic_vector(31 downto 0); -- AHBR Read Address
  Rdata : std_logic_vector(31 downto 0); -- AHBR Read data
  rd : std_ulogic; -- Read (ulogic for multiple concurrent signals)
  wr : std_ulogic; -- Write
  state : dcom_state_type; -- 'state' Type
  hresp : std_logic_vector(1 downto 0); -- Save HRESP Value
  Waddr : std_logic_vector(31 downto 0); -- AHBR Write Address
  Wdata : std_logic_vector(31 downto 0); -- AHBR Write Data
end record;
signal ahbReg, ahbIn : ahb_type;

begin
  rstn_inv <= not (rstn and apbreg.startJOP); -- Reset inverted for JOP with synch to start
end begin;

3) COMPONENT INSTANTIATION

jopcpu0 : jopcpu
  port map (clk => clkm, reset => reset_ft, sc_mem_out => jop_mem_out, sc_mem_in => jop_mem_in, sc_io_out => jop_io_out, sc_io_in => jop_io_in, irq_in => jop_irq_in, irq_out => jop_irq_out, exc_req => jop_exc_req);

ahbmst1 : ahbmst
  generic map (hindex, hirq, VENDOR_JOP, JOP_CORE)
  port map (rstn, clkm, dmai, dmao, ahbmi, ahbmo);

fault0 : fault
  port map (clock => clkm, reset => rstn_inv, enable => enable, spov => jop_exc_req.spov, ab => jop_exc_req.ab, np => jop_exc_req.np, reset_ft => reset_ft);

4A) BUS CONTROLLER: APB

apbctrl : process(rstn, apbReg, apbi, ahbReg)
  variable v : apb_type;
variable readdata : std_logic_vector(31 downto 0) := (others => '0');
variable xirq : std_logic_vector(NAHBRQ-1 downto 0) := (others => '0');
begin
  v := apbReg;
  -- Read Registers
  readdata := (others => '0');
  case apbi.paddr(6 downto 2) is
    when "00000" => readdata(31 downto 0) := v.startAddr;
    -- JOP Bootloader Addr
    when "00001" => readdata(31 downto 0) := v.outAddr;
    -- JOP Output Address
    when "00010" => readdata(0) := v.startJOP; -- JOP Start
    when "00011" => readdata(0) := dmao.active;
    -- Check DMAO Status
    readdata(1) := dmao.ready;
    readdata(2) := dmao.start;
    readdata(3) := jop_exc_req.spy; -- SP Overflow
    readdata(4) := jop_exc_req.np; -- Null Pointer
    readdata(5) := jop_exc_req.ab; -- Array Out of Bounds
  when others =>
  end case;
  -- Write Registers
  case apbi.paddr(6 downto 2) is
    when "00000" => v.startAddr := apbi.pwdata(31 downto 0);
    -- JOP Bootloader Addr
    when "00001" => v.outAddr := apbi.pwdata(31 downto 0);
    -- JOP Output Address
    when "00010" => v.startJOP := apbi.pwdata(0); -- JOP Start
    when others =>
  end case;
  -- Reset Operation
  if rstn = '0' then
    v.startAddr := (others => '0');
    v.outAddr := (others => '0');
    v.startJOP := '0';
  end if;
  apbIn <= v;
  apbo.prdata <= readdata;
  apbo.pirq <= xirq;
end process apbctrl;
apbo.pindex <= pindex;
apbo.pconfig <= pconfig;
regAPB: process(clkm)
begin
  if rising_edge(clkm) then
    apbReg <= apbIn;
  end if;
end process;
 -- Boot Message
-- pragma translate_off
-- pragma translate_on
-- 4B) BUS CONTROLLER: AHB
combin: process(dmao, rstn, ahbmi, ahbReg, apbReg, jop_io_out, jop_io_in,
jop_mem_out, jop_mem_in)
variable v : ahb_type;
variable vdmai : ahb_dma_in_type;
begin
  v := ahbReg;
  vdmai.start := '0';
vdmai.burst := '0';
vdmai.size := "00";
vdmai.busy := '0';
vdmai.address := ahbReg.addr;
vdmai.wdata := ahbReg.Wdata;
vdmai.write := ahbReg.wr;
vdmai.irq := '0'; -- Link with JOP IRQ?

-- Save hresp
if dmao.ready = '1' then v.hresp := ahbmi.hresp;
end if;

case ahbReg.state is
    -- Idle State
    when idle =>
        if apbReg.startJOP = '1' and rstn_inv = '0' then
            v.state := load;
        else
            v.state := idle;
        end if;
        v.wr := '0';
        v.rd := '0';
        v.addr := (others => '0');
        v.Raddr := (others => '0');
        v.Waddr := (others => '0');
    end when;

    -- Load Registers
    when load =>
        if jop_mem_out.rd = '1' or jop_io_out.rd = '1' then
            v.rd := '1';
            v.Raddr := apbReg.startAddr(31 downto 9) &
                       jop_io_out.address & "00";
            v.addr := v.Raddr;
            -- PROM Addr + JOP Req Addr
            v.Rdata := jop_mem_in.rd_data;
            v.state := read;
        else
            v.state := load;
            jop_mem_in.rdy_cnt(0) <= '1';
            jop_mem_in.rdy_cnt(1) <= '1';
            jop_io_in.rdy_cnt(0) <= '1';
            jop_io_in.rdy_cnt(1) <= '1';
        end if;
        if jop_mem_out.wr = '1' or jop_io_out.wr = '1' then
            v.wr := '1';
            v.Waddr := apbReg.outAddr(31 downto 9) &
                       jop_io_out.address & "00";
            v.addr := v.Waddr;
            v.Wdata := jop_mem_out.wr_data;
            v.state := write;
        end if;
    end when;

    -- AHB Read
    when read =>
        if dmao.active = '1' then -- Memory ready?
            if dmao.ready = '1' then
                jop_io_in.rd_data <= dmao.rdata;
                -- JOP read data < memory
                jop_mem_in.rd_data <= dmao.rdata;
                -- JOP read data < memory
                v.Rdata := dmao.rdata;
                jop_mem_in.rdy_cnt(0) <= '0';
                jop_mem_in.rdy_cnt(1) <= '0';
                jop_io_in.rdy_cnt(0) <= '0';
                jop_io_in.rdy_cnt(1) <= '0';
                v.rd := '0';
                v.state := load;
            else
                end if;
        end if;

end case;
v.rd := '1'; -- Next state = read
v.addr := v.Raddr;
v.state := read;
jop_mem_in.rdy_cnt(0) <= '1'; jop_mem_in.rdy_cnt(1) <= '1';
jop_io_in.rdy_cnt(0) <= '1'; jop_io_in.rdy_cnt(1) <= '1';
end if;
vdmai.start := '1'; -- Start Memory < AHB

-- AHB Write
when write =>
  if dmao.active = '1' then
    if dmao.ready = '1' then -- Memory ready?
      vdmai.wdata := v.Wdata; -- Mem < AHB write data
      v.wr := '1'; -- Next state = read
      v.addr := v.Waddr;
      v.state := load;
      end if;
    else
      v.wr := '1'; -- Next State = write
      v.addr := v.Waddr;
      v.state := write;
      jop_mem_in.rdy_cnt(0) <= '1'; jop_mem_in.rdy_cnt(1) <= '1';
      jop_io_in.rdy_cnt(0) <= '1'; jop_io_in.rdy_cnt(1) <= '1';
    end if;
    vdmai.start := '1'; -- Start Memory < AHBO
  end case;

if (reset_ft) = '1' then
  v.state := idle;
  v.rd := '0';
  v.wr := '0';
  v.addr := (others => '0');
  v.Raddr := (others => '0');
  v.Waddr := (others => '0');
  jop_mem_in.rdy_cnt(0) <= '1'; jop_mem_in.rdy_cnt(1) <= '1';
  jop_io_in.rdy_cnt(0) <= '1'; jop_io_in.rdy_cnt(1) <= '1';
end if;

ahbIn <= v;
dmai <= vdmai;
end process combin;

regAHB : process(clkm)
begin
  if rising_edge(clkm) then
    ahbReg <= ahbIn;
  end if;
end process;

end rtl;
This testbench was taken when the memory areas were set in ModelSim and generated using jopa.java to obtain the PROM, ROM, and RAM memory areas.

This simulation shows 3 key areas of interest:

1. An initialisation phase where stacks are yet to be fully set as the first bytecode fetch has not occurred. Only some sporadic reads are occurring here of initialisation addresses.

2. The fetching of the first method cache fill (around 20 micro-codes) into a bytecode array which is stored in the RAM cache.

3. The processor can then call each instruction from the ROM lookup table and run the sequences of codes to form and complete bytecodes.
Appendix F. System Properties Usable for Capability Function

-- listing properties --

java.runtime.name=Java(TM) 2 Runtime Environment, Standard...

sun.boot.library.path=C:\j2sdk1.4.2_12\jre\bin

jvm.version=1.4.2_12-b03
java.vm.vendor=Sun Microsystems Inc.
java.vendor.url=http://java.sun.com/
pattern.separator=;
java.vm.name=Java HotSpot(TM) Client VM
file.encoding.pkg=sun.io
user.country=GB

sun.os.patch.level=Service Pack 2
java.vm.specification.name=Java Virtual Machine Specification
user.dir=C:\eclipse\workspace\jade3.5
java.runtime.version=1.4.2_12-b03
java.awt.graphicsenv=sun.awt.Win32GraphicsEnvironment
java.endorsed.dirs=C:\j2sdk1.4.2_12\jre\lib\endorsed
os.arch=x86

java.io.tmpdir=C:\\DOCU\ME-1\eeplcb\LOCA\LS-1\Temp\
line.separator=
java.vm.specification.vendor=Sun Microsystems Inc.
user.variant=

os.name=Windows XP
sun.java2d.fontpath=
java.library.path=C:\j2sdk1.4.2_12\bin;.;C:\WINDOWS\sys...
java.specification.name=Java Platform API Specification
java.class.version=48.0

java.util.prefs.PreferencesFactory=java.util.prefs.WindowsPreferencesFac... os.version=5.1
user.home=C:\Documents and Settings\eeplcb
user.timezone=Europe/London
java.awt.printerjob=sun.awt.windows.WPrinterJob
file.encoding=Cp1252
java.specification.version=1.4
user.name=eeplcb
java.class.path=C:\eclipse\workspace\jade3.5\bin;C:\e...
java.vm.specification.version=1.0

200
Appendices

Hardware CPU Data Model

- sun.arch.data.model=32
- java.home=C:\j2sdk1.4.2_12\jre
- java.specification.vendor=Sun Microsystems Inc.
- user.language=en
- awt.toolkit=sun.awt.windows.WToolkit
- java.vm.info=mixed mode

Java Version

- java.version=1.4.2_12
- java.ext.dirs=C:\j2sdk1.4.2_12\jre\lib\ext
- sun.boot.class.path=C:\j2sdk1.4.2_12\jre\lib\rt.jar;C:\j2...
- java.vendor=Sun Microsystems Inc.
- file.separator=
- java.vendor.url.bug=http://java.sun.com/cgi-bin/bugreport...
- sun.cpu.endian=little
- sun.io.unicode.encoding=UnicodeLittle

Network IP Address + CPU Type

- java.rmi.server.hostname=169.254.33.126
- sun.cpu.isalist=pentium i486 i386

CPU: 2, freeMem: 1116360, maxMem: 2031616, totalMem: 66650112

No. CPUs + Mem. Avail.
Appendix G. S-Cube Picosatellite Design

G.1 Configuration, Structure, and Mass Budget

To fit the required functionality into one CubeSat platform the following configuration is recommended. Solar Cells would be placed on 4 panels with 1 panel for the Camera system/ground antenna/ISL patch antenna. ISL patch antennae could be placed on a number of panels along the tumbling plane to create a larger beam width. The camera board will also need 2 slots and the distributed computing board is limited to 1 slot.

The board configuration inside is as follows (from the bottom):

- MSP430 Flight Module as the main on-board computer (OBC)\textsuperscript{18}
- Power Board with 2 x Daughter Battery Boards\textsuperscript{19}
- Groundlink Communications Board\textsuperscript{20}
- Payload Board – FPGA/Camera Board\textsuperscript{21}

The current demonstrator design can be seen in Figure G-1.

\textbf{Figure G-1. CubeSat Platform with Flight Module, Power Board, communications board, and FPGA/Camera Board (with and without structure)}

\textsuperscript{21} R. Barnes and T. Vladimirova, “CubeSat Remote Imaging Subsystem – Final Report”, MSc Thesis, University of Surrey, 7\textsuperscript{th} May 2008
Appendices

Figure G-1 shows the current demonstrator with the Flight Module, the power board, the communications board, and FPGA/Camera Board attached with and without the structure.

Table G-1 shows the mass budget for the proposed S-Cube picosatellite design. This is used to ensure that circuit boards not only fit into the required structure but also the mass does not exceed the 1 kg limit.

Table G-1. Mass Budget for S-Cube

<table>
<thead>
<tr>
<th>Board</th>
<th>Size (mm)</th>
<th>Mass (g)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Structure</td>
<td>n/a</td>
<td>113.5</td>
</tr>
<tr>
<td>Solar Panels x 4</td>
<td>92 x 92 x (3)</td>
<td>c. 220</td>
</tr>
<tr>
<td>FM430 OBC</td>
<td>96 x 90 x 11.4</td>
<td>74</td>
</tr>
<tr>
<td>Clyde-Space EPS Board</td>
<td>95 x 90 x (15)</td>
<td>80</td>
</tr>
<tr>
<td>+ Clyde-Space Battery Daughter Boards</td>
<td>95 x 90 x 21</td>
<td>124</td>
</tr>
<tr>
<td>Ground Link Communications Board</td>
<td>96 x 58 x 14.5</td>
<td>c. 140</td>
</tr>
<tr>
<td>Payload: IEEE 802.11 ISL Radio+ Distributed Computing + Camera Board</td>
<td>95 x 95 x 27</td>
<td>c. 110</td>
</tr>
<tr>
<td>Antenna</td>
<td>92 x 92 x 6</td>
<td>c. 130</td>
</tr>
<tr>
<td>Cabling</td>
<td>n/a (PC/104)</td>
<td>0</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>Height: 93.9</strong></td>
<td><strong>991.5</strong></td>
</tr>
</tbody>
</table>

G.2 Orbital Considerations

For this mission, a number of future launches have been considered. These include launches as a piggy back with SSTL on a Dnepr launcher 4, ESA’s Vega launcher and two Indian PSLV launches.

These launch provisions have the following orbit injection parameters:

- Dnepr Launch (SSTL) – Q4 2009: 686 km Sun Synchronous
- ESA Vega Launch 1 – Q4 2009/ Q1 2010: 1200 x 350 km (71° inclination)
- ESA Vega Launch 2 – Q4 2009/ Q1 2010: 350 x 350 km (71° inclination)
- PSLV Launch 1 – June 2009: c. 817 km Sun Synchronous
- PSLV Launch 2 – July/ September 2009: c. 670 km Sun Synchronous

Some of these launches are not suitable for a number of reasons. Table G-2 summarises why some future launches cannot be considered.

Table G-2. Orbit Comparison and Project Readiness

<table>
<thead>
<tr>
<th>Launcher</th>
<th>Launch Time</th>
<th>Altitude (km)</th>
<th>Inclination (°)</th>
<th>Suitability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dnepr</td>
<td>Q4 2008</td>
<td>686</td>
<td>98</td>
<td>Too soon</td>
</tr>
<tr>
<td>Vega Launch 1</td>
<td>Q4 2008/Q1 2009</td>
<td>1200 x 350</td>
<td>71</td>
<td>Too soon, highly elliptical</td>
</tr>
<tr>
<td>Vega Launch 2</td>
<td>Q4 2008/Q1 2009</td>
<td>350</td>
<td>71</td>
<td>Too soon, (too low)</td>
</tr>
<tr>
<td>PSLV Launch 1</td>
<td>June 2009</td>
<td>817</td>
<td>98</td>
<td>(Too high for G/L range)</td>
</tr>
<tr>
<td>PSLV Launch 2</td>
<td>July/Sept. 2009</td>
<td>670</td>
<td>98</td>
<td>OK</td>
</tr>
</tbody>
</table>

For a good long-lived mission, a sun-synchronous orbit to a maximum of 800 km is to be considered and from Table G-2, it can be seen that only the PSLV Launch 2 offers us this opportunity. With either of these launches, the use of a P-POD could be used to deploy 3 picosatellites based on the CubeSat standard in a walker scenario in a ‘slave’ – ‘master’ – ‘slave’ configuration as shown in Figure G-2. For a sun-synchronous orbit, an inclination of 98° is assumed.

Figure G-2. Proposed S-Cube Mission

G.3 Power System Estimation & Eclipse Times

For the power system, the orbital period and time in eclipse must be found. Assuming a circular low Earth orbit (LEO), these are represented in Equations 11, 12 and 13 respectively:

\[
P = 2\pi \left( \frac{a^3}{\mu_G} \right)
\]  

(G-1)

Where \( P = \) the orbital period, \( a = \) semi-major axis and \( \mu = \) the standard gravitational parameter \((G, \text{gravitational constant, time } M, \text{the mass}); \) a constant of \(3.986 \times 10^5\).
\[ T_E = \frac{2\rho}{360^\circ} \]  

(G-2)

Where \( \rho = \sin^{-1}\left(\frac{R_{\oplus}}{R_{\oplus}+H}\right) \)  

(G-3)

Where \( T_E \) = time in eclipse, \( P \) = orbital period, \( \rho \) = the Earth’s angular radius, \( R \) = Earth’s radius and \( H \) = satellite altitude.

For PSLV Launch 2:

Orbital Period, \( P = 2\pi \left(\frac{(6378.14 + 670)^3}{3.986 \times 10^5}\right) \frac{5.9088 \times 10^3}{60} = 98.1458 \) minutes

Earth’s angular radius, \( \rho = \sin^{-1}\left(\frac{R_{\oplus}}{R_{\oplus}+H}\right) = \sin^{-1}\left(\frac{6378.14}{6378.14 + 670}\right) = 64.82^\circ \)

Time in eclipse, \( T_E = \frac{2\rho}{360^\circ} = 98.1458 \times \left(\frac{2 \times 64.82}{360}\right) = 35.34 \) minutes

**G.4 Power Budget**

A standard power budget table is presented in Table G-3 as defined in Space Mission Analysis and Design. This method is the most commonly used power budget system as it incorporates a conservative and failure safe method against worst case scenarios. The values are taken from current power requirements onboard the CubeSat and possible duty cycles of the satellite in a sun-synchronous orbit. Duty cycle values are taken based on the length of time before damage is taken to the battery system by exceeding the maximum depth of discharge (DOD).

<table>
<thead>
<tr>
<th>Sub-System</th>
<th>Active (mW)</th>
<th>Sunlit Duty Cycle (%)</th>
<th>Sunlit Power Req't (mW)</th>
<th>Power Down (mW)</th>
<th>Eclipse Duty Cycle (%)</th>
<th>Eclipse Power Req't (mW)</th>
</tr>
</thead>
<tbody>
<tr>
<td>OBC (FM430)</td>
<td>66</td>
<td>100</td>
<td>66</td>
<td>22</td>
<td>100</td>
<td>22</td>
</tr>
<tr>
<td>Communication (UHF Tx)</td>
<td>110</td>
<td>20</td>
<td>220</td>
<td>70.3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Communication (VHF Rx)</td>
<td>259</td>
<td>10</td>
<td>25.9</td>
<td>70.3</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

 Payload: IEEE 802.11 ISL Radio\textsuperscript{25} & 1000 & 10 & 100 & 0 & 0 & 0 \\
 Payload: Distributed Computing & 1022 & 20 & 204.4 & 0 & 0 & 0 \\
 - SDRAM (MT48LC32M16A2) & 360 & & & & & \\
 - FPGA (XC5VLX50-3ff324)\textsuperscript{26} & 632 & & & & & \\
 - PROM (XCF32PVO48) & 30 & & & & & \\
 Payload 2 Camera & FPGA \textsuperscript{21} & 400 & 5 & 20 & 0 & 0 & 0 \\
 Attitude Determination & Control & N/A & 0 & & & \\
 Power Board (Clyde Space)\textsuperscript{27} & 100 & 100 & 100 & 0 & 100 & 100 \\
 Thermal & Structure & N/A & 0 & & & \\
\textbf{Total Required} & 2957 & 736.3 & 122 & & & \\

Key power budget values:

- **Total Power Requirement (Active):** 2957 mW – Must be met by the EPS System as a fail-safe measure if there is a programming error, a single-event upset, or other systems failure on board the satellite.

- **Sunlit Power Requirement:** 736.3 mW – Must be met by the solar arrays during sunlit orbit time.

- **Eclipse Power Requirement:** 122 mW – Must be met by the batteries during eclipse orbit time and prevent the maximum DOD from being reached.

G.5 Solar Panel and Batteries

The solar panels are single junction 2 x 4 cm GaAs/Ge cells rated at 18% efficiency (860 mV, 25%, AM0). With two parallel strings of four cells in series, this translates to 1574 mW (457 mA @ 3.44V). Halving this for a 200% margin gives a result of 787 mW on average which meets the sunlit power requirement of 736.3 mW. Compared to the total sunlit power requirement, this is still in range. This design is shown in Figure G-3.

\textsuperscript{25} IEEE Computer Society, “IEEE Std 802.11\textsuperscript{TM} 2007”, Section I.2.2. Transmit Power Levels, pp. 1143

\textsuperscript{26} Taken from XPower Measurements of LEON3 + JOP on a Virtex-5 LX50 (40 MHz, 12.5 toggle rate)

\textsuperscript{27} Clyde Space, “CubeSat Power System User Manual”, Issue L, 23\textsuperscript{rd} July 2008
The Clyde-Space daughter battery board contains 2 cells and is rated at 10 W/hr or 1.25 Ah at 8.2V but to check the required capacity of the batteries Equation (G-4) is used:

\[
C_r = \frac{P_e T_e}{(DOD)Nn} \text{ W/hr}
\]  

(G-4)

Where \(P_e\) = power eclipse, \(T_e\) = time eclipse, DOD = depth of discharge, \(N\) = no. of batteries, \(n\) = transmission efficiency between EPS system and load.

From Clyde Space’s EPS board and daughter battery board, the following characteristics are used: 

\(P_e = 0.122\text{ W}, \ T_e = 35.34\text{ minutes}, \ DOD = 30\%, \ N = 2\text{ and }n = 0.90.\)

For PSLV Launch 2: 

\[
C_r = \frac{0.122 \times 35.34}{(0.30) \times 2 \times 0.90} = 7.984 \text{ W/hr (In range of the 10 W/hr margin)}
\]
Appendix H. S-Cube CAD Drawings
Appendix I. JADE-LEAP-pJava Verbose Output

```
$ make jsim
rm -rf java/target/dist
mkdir java/target/dist
mkdir java/target/dist/classes
mkdir java/target/dist/lib
mkdir java/target/dist/bin
javac -d java/target/dist/classes -sourcepath java/target/src/common\java\target/src/jdk\_base\java\target/src/jdk11\c;eclipse/rt\extras\java\target/src/rt
p\java\target\src\examples\java\target\src\app\java\target\src\bench -bootclasspath "" -extdirs "" -classpath "java/target\jade\classes" -source 1.4 java\target\src\common\com\jopdesign\sys\*.java
javac -d java/target/dist/classes -sourcepath java/target/src/common\java\target\src/jdk\_base\java\target\src/jdk11\c;eclipse/rt\extras\java\target\src/rt
p\java\target\src\examples\java\target\src\app\java\target\src\bench -bootclasspath "" -extdirs "" -classpath "java/target\jade\classes" -source 1.4 java\target\src\common\com\jopdesign\sys\*.java
javac -d java/target/dist/classes -sourcepath java/target/src/common\java\target\src/jdk\_base\java\target\src/jdk11\c;eclipse/rt\extras\java\target\src/rt
p\java\target\src\examples\java\target\src\app\java\target\src\bench -bootclasspath "" -extdirs "" -classpath "java/target\jade\classes" -source 1.4 java\target\src\common\com\jopdesign\sys\*.java
javac -d java/target/dist/classes -sourcepath java/target\src\common\jav\target\src/jdk\_base\java\target\src/jdk11\c;eclipse/rt\extras\java\target\src/rt
p\java\target\src\examples\java\target\src\app\java\target\src\bench -bootclasspath "" -extdirs "" -classpath "java/target\jade\classes" -source 1.4 java\target\src\common\com\jopdesign\sys\*.java
```

Inclusion of:
- jdk1.1.8 source
- phoneME source
- JADE source

All the new class files loaded to run JADE-LEAP-pJava
Appendices
null

Exception in thread "main" java.lang.StackOverflowError
  at org.apache.regexp.RECompiler.node(Unknown Source)
  at org.apache.regexp.RECompiler.atom(Unknown Source)
  at org.apache.regexp.RECompiler.terminal(Unknown Source)
  at org.apache.regexp.RECompiler.closure(Unknown Source)
  at org.apache.regexp.RECompiler.branch(Unknown Source)
  at org.apache.regexp.RECompiler.expr(Unknown Source)
  at org.apache.regexp.RECompiler.terminal(Unknown Source)
  at org.apache.regexp.RECompiler.closure(Unknown Source)
  at org.apache.regexp.RECompiler.branch(Unknown Source)
  at org.apache.regexp.RECompiler.expr(Unknown Source)
  at org.apache.regexp.RECompiler.compile(Unknown Source)
  at org.apache.regexp.RE.<init>(Unknown Source)
  at org.apache.regexp.RE.<init>(Unknown Source)
  at com.jopdesign.build.ClinitOrder.findDependencies(ClinitOrder.java:68)
  at com.jopdesign.build.ClinitOrder.findDependencies(ClinitOrder.java:89)
  at com.jopdesign.build.ClinitOrder.findDependencies(ClinitOrder.java:119)
  at com.jopdesign.build.ClinitOrder.findDependencies(ClinitOrder.java:119)
  at com.jopdesign.build.ClinitOrder.findDependencies(ClinitOrder.java:119)
  at com.jopdesign.build.ClinitOrder.findDependencies(ClinitOrder.java:119)
  at com.jopdesign.build.ClinitOrder.findDependencies(ClinitOrder.java:119)
  at com.jopdesign.build.ClinitOrder.findDependencies(ClinitOrder.java:119)
  at com.jopdesign.build.ClinitOrder.findDependencies(ClinitOrder.java:119)
  at com.jopdesign.build.ClinitOrder.findDependencies(ClinitOrder.java:119)
make: *** [java_app] Interrupt