Auton Agent Multi-Agent Syst (2018) 32:252–274
https://doi.org/10.1007/s10458-017-9378-4

A hybrid real-time agent platform for fault-tolerant, embedded applications

A. O. Erlank[1] · C. P. Bridges[1]

Published online: 17 November 2017

Abstract This paper describes an agent platform based on the Foundation for Intelligent Physical Systems Abstract Architecture, which, together with a highly fault tolerant, bio-inspired hardware architecture, aims to increase the reliability of future, low-cost satellites. To achieve the stringent operational requirements imposed by the real-time and resource-constrained environment of a satellite, the Hybrid Agent Real-Time Platform (HARP) distinguishes itself from other platforms in three areas. Firstly, the HARP middleware uses discrete processors, instead of virtual machines or interpreters, as its agent execution environment. This has the advantage of reducing the agency memory footprint and enabling agents to perform real-time tasks. Secondly, the HARP communication stack makes use of ISO-TP over CAN 2.0A as its transfer level protocol, cutting out resource-intensive layers such as HTTP and IIOP. In addition, the communication stack allows real-time CAN traffic to share the network and be given priority over Agent Communication Language messages. Finally, the HARP middleware embeds a peer-to-peer task manager in each agency, allowing systems which are built using the bio-inspired Artificial Stem Cell Architecture and HARP middleware to autonomously reconfigure in the event of failures. The detailed design of the HARP middleware is given, together with details of an implementation of the HARP middleware on a set of prototype satellite hardware. The performance and scaling potential of the middleware, determined through a set of physical experiments, provide evidence of the practical feasibility of the proposed architecture.

Keywords Fault tolerance · Multi-agent systems · Real-time · Satellites · Bio-inspired

✉ A. O. Erlank
a.erlank@surrey.ac.uk

C. P. Bridges
c.p.bridges@surrey.ac.uk

1 Surrey Space Centre, BA Building, University of Surrey, Guildford, United Kingdom

1 Introduction

A new class of small, low-cost satellites is enabling novel, distributed mission concepts. These satellites are rapidly growing in popularity and capability, but have shown poor on-orbit reliability to date [1]. Since traditional techniques for improving reliability are largely incompatible with the intrinsic limitations on mass, volume and power of this class of satellite, alternative techniques are required. To this end, the Satellite Stem Cell distributed architecture, inspired by multicellular life and based on the concept of reconfigurable artificial cells, has been proposed [1,2].

To facilitate communication and cooperation between these cells, which are based on distributed microcontrollers (MCUs), a set of middleware is required. The middleware is responsible for keeping the system operational, even in the event of partial hardware failures. To this end, it must transparently handle inter-cellular communication, task distribution and health monitoring. In addition, the middleware must be compatible with the resource-constrained, real-time environment of a small, low-cost satellite. While bespoke solutions are popular in the space industry, a solution based on existing and popular technologies better fits the ethos of these missions.

The Common Object Request Broker Architecture (CORBA) is a popular middleware architecture designed to allow applications coded in different languages and running on different hardware, operating systems and networks, to interact. CORBA allows an application to be a client and a server simultaneously and facilitates the communal manipulation of software objects. While CORBA is often cited as being overly complex and having a large implementation footprint [3], several reduced versions have been created for embedded and real-time applications [4–6]. Nevertheless, CORBA is over twenty years old and its overall use is in decline [7].

Alternatives to CORBA can be found in the domain of wireless sensor networks (WSNs). These are networks of small, low-power embedded systems, typically designed for environmental monitoring. A variety of middleware, focusing on reconfigurability, scalability, low power consumption and real-world integration, has emerged from research into this field [8]. Popular examples include Mate [9], which turns each node in a network into a custom byte code interpreter, and TinyDB [10], which allows a whole network of nodes to be queried like a single database.

Closely related to WSNs is the concept of the Internet-of-Things (IoT), which aims to to interconnect small, everyday, embedded devices through the use of web technologies and lightweight data-interchange formats such as JSON. A good survey of middleware developed for IoT is given in [11].

While communication and cooperation between the cells of the Satellite Stem Cell architecture could be enabled through middleware based on CORBA, or adapted from WSNs or IoT, the field of agent technology provides a more conceptually natural solution. Similar to the cells in a multicellular organism, agents have life cycles, pursue goals and may have a level of mobility. Thus, this paper describes an embedded agent platform, named Hybrid Agent Real-Time Platform (HARP), which was developed to enable end-user tasks to be efficiently and reliably executed on the reconfigurable cell hardware.

Agent platforms can be categorised according to their target hardware platform, their real-time capabilities, and their Foundation for Intelligent Physical Agents (FIPA) compliance [12]. Target hardware platforms range from desktop computers with multi-gigahertz processors and gigabytes of RAM to small, embedded, MCU-based platforms running at a few megahertz and with a few kilobytes of RAM. An agent platform's real-time capabilities depend on its agent execution environment, which may be based on a runtime interpreter, a virtual machine, or native hardware. Finally, FIPA compliance means an agent platform adheres to a set of standards governing heterogeneous agent interaction. FIPA-compliant agents and platforms may form part of a larger agent ecosystem, instead of being confined to their respective platforms.

The most common agent platforms are those designed to operate on standard personal computer hardware. These platforms are typically used for multi-agent-based simulations applied to a variety of research fields, including economics, social sciences, biology and urban planning. Since these applications do not have real-time requirements, agents are typically coded in Java and execute within Java Virtual Machines (JVMs). A good survey of 24 such platforms can be found in [13], with popular, FIPA-compliant examples including JADE [14,15] and FIPA-OS [16]. While these platforms are powerful and multifunctional, their agent execution timing is non-deterministic due to the underlying JVM's garbage collection routines, class initialisation, and just-in-time compilation [17]. In addition, their large memory requirements make them unsuitable for implementation in the resource-restricted, embedded environment of a satellite.

Micro-agents were proposed in response to the efficiency penalties imposed by fully-fledged agent environments when deployed in production environments. They compromise on expressiveness in favour of performance and scalability, and are a proposed middle-ground between fully fledged FIPA agents and typical programming frameworks and data structures. A survey of Java-based, multi-agent platforms, supporting micro-agents, is given in [18].

Agent platforms with reduced memory requirements exist for devices such as smart phones. Examples include u-FIPA [19], JADE-LEAP [20–22] and AgentLight [23]. These platforms typically make use of the Java ME Embedded or Android run-time environments. The Java ME Embedded runtime environment is available for many platforms, including ARM-Cortex M3 MCUs and requires as little as 128 kB of RAM and 1MB of ROM [24]. AgentLight has demonstrated the creation of ten agents using only 83 KB of memory.

Mobile-C was developed to have a small memory footprint and be FIPA compliant, but differs from the majority of agent platforms by using an embedded C/C++ interpreter as its agent execution environment [25]. The embedded interpreter, named *Ch*, is claimed to be smaller than the Java ME Embedded runtime environment. The C language was chosen to allow tight coupling to low-level hardware, which is often a requirement in embedded applications.

Even smaller agent platforms are available from the distributed sensor node community. For example, Agilla requires only 57 kB of ROM and 3.3 KB of RAM when implemented on an ATmega128L MCU [26]. Agilla agents are written in a custom language and execute in a custom virtual machine. Agilla is not FIPA compliant.

While agent platforms based on the Java ME Embedded runtime environment, Mobile-C and Agilla would all fit within the processing and memory restrictions of a satellite embedded system, their interpreted agent execution environments make them unsuited for real-time tasks such as attitude determination and control.

One approach to overcoming the timing problems of interpreted agent execution environments is to replace the JVM with a hardware Java processor. With the aim of developing a real-time agent platform for satellites, Bridges et. al. attempted to get the JADE-LEAP mobile agent platform operating on a Java Optimised Processor (JOP) [27]. The JOP was implemented in a Xilinx Spartan-3 field-programmable gate array (FPGA). While the feasibility of this concept was proven, a useable platform could not be developed. Zabel et al. have developed the Secure Hardware Agent Platform (SHAP) Java processor for enabling a real-time agent platform [28]. It has been demonstrated with the Connected Limited Device

Configuration (CLDC) API, which is a subset of the Java ME Embedded runtime environment. However, as far as the author is aware, no agent platform has yet been adapted or developed to make use of this processor.

While there are several challenges to getting agent technology applied on board satellites, at least one agent has seen on-orbit operation. In 1999, NASA briefly handed control of its Deep Space 1 probe to an experimental on-board agent [29]. The agent operated as top level software and did not execute real-time processes. More recently, Princeton Satellite Systems developed the ObjectAgent environment for its Techsat 21 mission [30]. However, this platform is designed to run on a powerful PowerPC 750 processor, is not FIPA compliant, and was never completed as the Tech-Sat mission 21 was cancelled.

The agent platform introduced in this paper, HARP, is designed to make best use of the unique, but constrained, hardware of the Satellite Stem Cell Architecture. It is characterised by a small memory footprint, real-time capabilities, and extensive fault tolerance. These features are achieved by departing from typical agent platform design in three areas. Firstly, the HARP middleware does not make use of an interpreter or virtual machine, and instead makes use of a set of discrete MCUs as its agent execution environment. Secondly, the agent communication channel supports both complex agent messages and real-time traffic, by using Controller Area Network (CAN) buses and cutting out complex application level network protocols. Finally, an embedded, distributed task allocation strategy allows all agencies in the system to react by automatically creating or migrating agents in response to changing conditions and failures. In addition, the HARP middleware allows the agencies themselves to be automatically moved in the event of hardware failures.

The HARP middleware is a hybrid system, as it allows a combination of high-level and real-time agent-based solutions to be implemented. Traditional agent-based solutions for high-level tasks, such as payload usage planning, can make use of the FIPA Agent Communication Language (ACL) and the services offered by the FIPA Abstract Architecture agency. On the other hand, real-time agents executing as native code on the discrete, MCU-based execution environment can make use of ACL for initial configuration and handshaking, before switching to traditional, deterministic, CAN messaging.

The rest of this paper is structured as follows. Section 2 gives an overview of the Satellite Stem Cell Architecture, including its aims, composition and features. In Sect. 3 the Hybrid Agent Real-Time Platform is described in detail. An implementation of the HARP middleware on prototype Satellite Stem Cell hardware is described in Sect. 4, together with experimental results and performance benchmarks. Section 5 gives a brief case study focussed on 'cellularising' a set of microsatellite avionics, before Sect. 6 concludes the paper.

2 The satellite stem cell architecture

The Satellite Stem Cell Architecture was developed as a proposed solution to the problem of low reliability seen to date amongst small, low cost satellites [1,2]. It proposes that satellites can be made out of a set of initially identical, reconfigurable hardware blocks, instead of unique, discrete subsystems. The initially identical hardware blocks will reduce manufacturing costs and potentially simplify testing and integration, while giving the system the intrinsic ability to reconfigure after failures.

This architecture was inspired by multicellular, biological life. In a simplified sense, multicellular organisms start out as a set of initially identical cells called stem cells. As the organism develops, these cells adapt to perform different roles through a process known

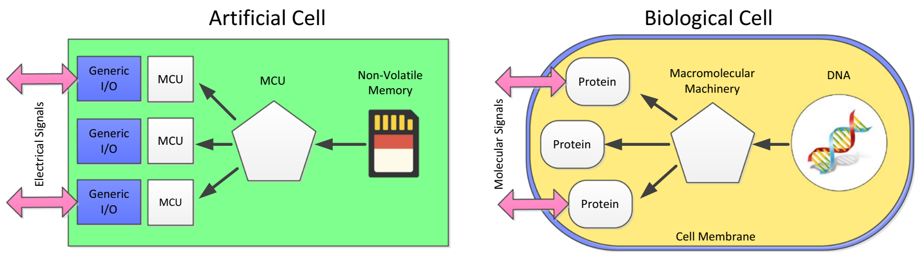


Fig. 1 A simplified schematic representation of the differentiation process in biological and proposed artificial cells. Adapted from [1]

as differentiation. Once the organism is fully developed, some cells have the ability to re-differentiate to take on new roles, often in response to damage to the organism. For example, damage to the lens cells in a newt's eye will trigger cells in the area to re-differentiate into lens cells to replace the lost functionality. Similarly, a Zebra fish can survive losing up to 20% of its heart thanks to the re-differentiation and proliferation of surviving cells [31]. Since man-made systems do not yet have the ability to self-proliferate, we must be content with being able to sacrifice some system performance in exchange for keeping vital tasks running. This is called graceful degradation and is a major aim of the Satellite Stem Cell Architecture.

The biggest challenge of achieving intrinsic graceful degradation in a system is designing hardware blocks which are sufficiently capable to perform any task required in the system. Traditional subsystem design is equivalent to designing fully differentiated, or specialised, cells. Each subsystem is designed to perform only a certain set of tasks. To design an artificial stem cell requires either building a subsystem which contains all the hardware of the other subsystems, or a subsystem which is reconfigurable enough to be able to do everything. The first option is undesirable due to volume and mass restrictions on board small satellites. Thus, the Satellite Stem Cell Architecture is based on artificial stem cells which mimic the reconfigurability of biological cells.

Figure 1 depicts a simplified representation of the differentiation process in a biological cell. Every cell in an organism contains an identical set of blueprints, called DNA, for building proteins. A set of complex molecules within the cell, called macromolecular machinery, respond to internal and external conditions by reading different sections of the DNA and building the corresponding proteins. It is these proteins which are ultimately responsible for performing the majority of cellular tasks. Thus, changing conditions inside or outside a cell can cause changes to the set of proteins currently in the cell. This set of proteins can be thought of as a set of tools which determine the cell's current capabilities and hence its specialisation.

Figure 1 also shows the cell membrane, which acts as an interface between the proteins and the outside world, while also protecting the cell's internals.

The Satellite Stem Cell Architecture is based on artificial cells which mimic the design of biological cells, as shown in Fig. 1. In these artificial cells, tasks are performed by artificial proteins, based on small processing elements such as MCUs. These processing elements can be considered blank proteins, which get loaded with firmware, read from non-volatile memory, by a processing element performing the role of macromolecular machinery (MM). The MM processor responds to changing internal and external conditions by loading different sets of firmware onto its set of artificial proteins. Each artificial protein also has a set of generic input/output (I/O) circuitry, which enables it to interface to a wide variety of extra-cellular peripherals, while being protected from unexpected external events.

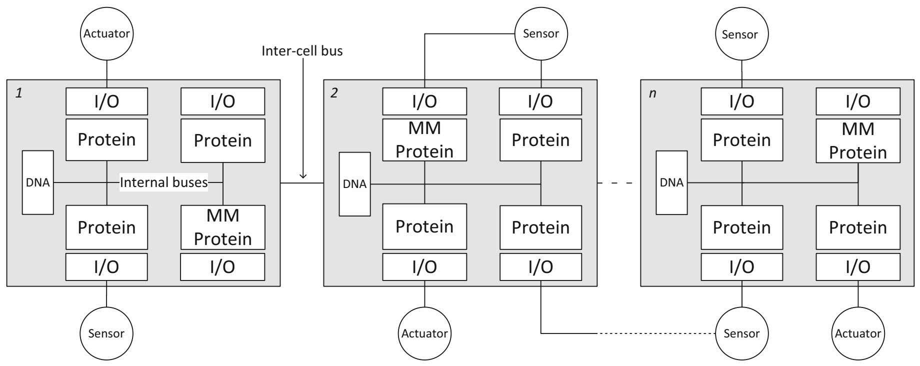


Fig. 2 A schematic representation of a system based on artificial cells. At any point in time, one protein per cell has the role of the cell's macromolecular machinery (MM). Peripherals interface to proteins via general purpose I/O circuitry and can be cross-strapped between proteins, or between cells. Reproduced from [2]

The macromolecular machinery may appear to be a potential single point of failure. However, in biological cells, the macromolecular machinery is itself composed of proteins. Therefore, in the artificial cell, the macromolecular machinery can be implemented in the exact same way as the artificial proteins. In this way, any of the proteins can perform the role of macromolecular machinery. Thus, if an individual protein is damaged, the macromolecular machinery will respond by redistributing the lost task to a protein which is currently performing a lower priority task. If the macromolecular machinery protein fails, one of the remaining proteins will take over from it.

An artificial multicellular system is created when a number of cells are linked together, as shown in Fig. 2. At this point the cells are required to cooperate to ensure that multiple cells are not unnecessarily performing the same tasks, and that peripherals are correctly shared.

In complex biological multicellular organisms, many bodily functions are coordinated centrally, through a nervous system and brain. Similarly, traditional satellite system design has a central on-board computer controlling the rest of the subsystems. A centrally-coordinated architecture is good for processing and memory efficiency, as a globally optimal task allocation strategy can be employed. Furthermore, the distributed processors can be fully devoted to task execution, without having to contain the intelligence for coordinating a distributed task allocation strategy. However, the central coordinating authority in this design represents a potential single point of failure.

One the other extreme are fully decentralised architectures where all task allocation is handled through peer-to-peer interactions. Such architectures have the advantage of being very robust to failures, but the distributed task allocation strategy causes memory and processing overheads for each of the distributed processors.

The Satellite Stem Cell architecture represents a compromise between these two architectures. It mimics the two-tiered architecture of simple multicellular organisms such as jellyfish, which have no central brains. Task allocation is coordinated in a peer-to-peer fashion at a cellular level, while task execution occurs at the protein level. Thus, it does not have a potential single point of failure, while also leaving its processing elements (proteins) unburdened from system-level decision making. What distinguishes the Satellite Stem Cell architecture from other multi-tiered architectures is the blurry line between the tiers. Since the MMs, which represent the top tier, are composed of the same hardware as the proteins (bottom tier), the system can continuously reconfigure in the event of failures to ensure the two-tiered architecture remains functional. Furthermore, the Satellite Stem Cell architecture represents more than a computational system, as every protein has its own generic I/O hardware, allowing complex sensor/actuator systems to be developed without additional electronics.

To facilitate the inter-cellular communication and decentralised task management on a system composed of artificial cells, middleware is required. The developed solution is based on the concept of Agent Computing, and is described in detail in Sect. 3.

3 The hybrid agent real-time platform

Agent-based computing emerged in the 1990s and has since become a powerful new programming paradigm [25]. Essentially, agent-based software development allows complex problems or systems to be broken down into high-level abstractions. Most commonly, agents are implemented as pieces of software which execute within a software framework called an agency.

The HARP middleware turns every artificial cell into an agency capable of executing multiple protein agents simultaneously. As expected from an agent platform, the HARP middleware agencies provide a variety of services which allow protein agents to interact with each other and their host cell agencies. In addition to these standard tasks, however, the cell agencies cooperate directly to ensure that a sufficient spread of proteins exist within the system at any one time to keep the system functional. The agencies monitor the health of their protein agents and each other, and respond to failures cooperatively. In the hopes of providing compatibility with future agent-based satellites and ground facilities, the HARP middleware is based on the FIPA Abstract Architecture.

3.1 FIPA abstract architecture overview

The FIPA Abstract Architecture was created to allow the interaction of agents from different platforms, without restricting platform developers to a single implementation language or platform hardware. Thus, the FIPA specifications define the basic components required of any compatible platform, as well as how agents interact with those components and each other. The essential components of a FIPA platform include an Agent Execution Environment (AEE), Agent Management System (AMS) and Agent Communication Channel (ACC). The design of each of these core components will be described, along with extended platform features relating to peer-to-peer task allocation, fault tolerance and real-time operations. Schematic representations of both the FIPA Abstract Architecture and the HARP middleware are shown in Fig. 3, and will be referred to throughout this section.

3.2 Agent execution environment (AEE)

The Agent Execution Environment is at the core of an agent platform. It is responsible for running agent code on the host machine and is often designed to be platform independent to allow agent mobility across heterogeneous platforms. In addition, it is responsible for enacting a security policy to ensure that agents do not have inappropriate access to the host machine. AEEs are commonly implemented using virtual environments, interpreters or virtual machines. For example, the JADE platform makes use of a Java virtual machine, while Mobile-C uses a C interpreter.

Due to the limited processing resources and real-time requirements of embedded satellite systems, the HARP middleware does not make use of virtual machines or interpreters.

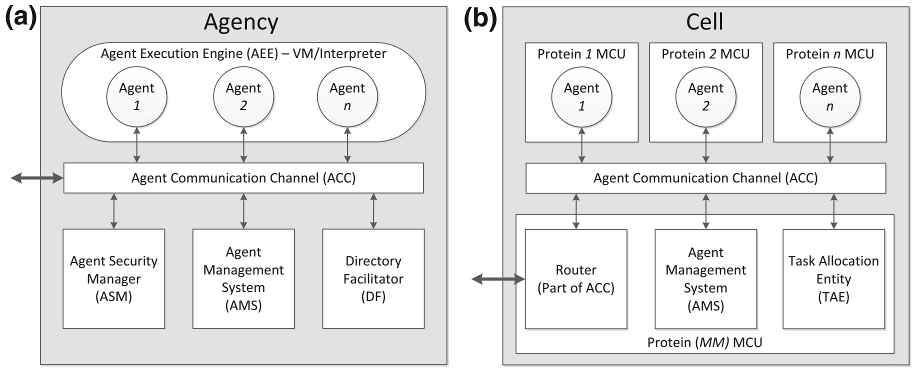

(a)
Agency
Agent Execution Engine (AEE) – VM/Interpreter
Agent 1
Agent 2
Agent n
Agent Communication Channel (ACC)
Agent Security Manager (ASM)
Agent Management System (AMS)
Directory Facilitator (DF)
(b)
Cell
Protein 1 MCU
Protein 2 MCU
Protein n MCU
Agent 1
Agent 2
Agent n
Agent Communication Channel (ACC)
Router (Part of ACC)
Agent Management System (AMS)
Task Allocation Entity (TAE)
Protein (MM) MCU

Fig. 3 Schematic representations of an agent platform based on the FIPA Abstract Architecture (**a**), and an agent platform based on the HARP middleware running on an artificial Satellite Stem Cell (**b**). Note that the Directory Facilitator is an optional component which is not currently implemented in the HARP middleware. Modified from [2]

Instead, it executes agents as native code on discrete processors. As described in Sect. 2, the hardware on which the HARP middleware resides is composed of a number of discrete processors. At any one time, one processor is reserved for executing agency services, but the remaining processors are used to execute agents. This is highlighted in Fig. 3b. Agents must be precompiled for the specific processor hardware before they can be deployed onto the system. Thus, agent mobility between platforms based on different processors is not supported. However, agent mobility within the same system is supported. Each processor contains a bootloader, which allows its memory to be reprogrammed with the compiled agent code. This process is orchestrated by the Agent Management System.

3.3 Agent management system (AMS)

The Agent Management System is responsible for managing the lifecycles of agents. It resides, along with other services, on the macromolecular machinery protein. According to FIPA, the AMS is responsible for agent creation, registration, deletion and mobility. To create a new agent, the AMS reads compiled agent code from its cell's central non-volatile memory and writes it into the memory of a protein processor. This process mimics the biological process of macromolecular machinery synthesizing proteins from instructions stored in DNA. Every agent has a unique, hard-coded identifier, which it uses to register with the AMS. In addition, the AMS stores the location of the agent (which processor it is executing on), which translates into a unique network address. The AMS maintains a list of all agents currently executing in its agency, or cell. This list can be queried by an agent to find the address of another agent on the cell, or via inter-agency messaging, an agent on another cell.

3.4 Agent communication channel (ACC)

Inter-agent communication is facilitated by an Agent Communication Channel. According to FIPA, the ACC is an entity responsible for transferring Agent Communication Language messages between agents. It does so by interacting with other agency services, such as the AMS agent directory, to provide a Message Transport Service (MTS). Since the goal of effective inter-agent communication lies at the heart of the FIPA ideology, FIPA supplies

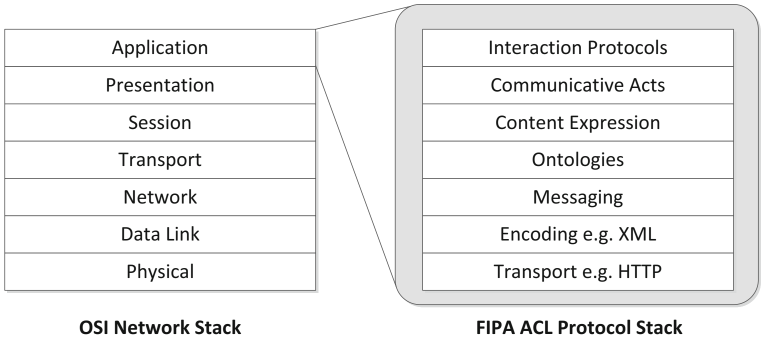


Fig. 4 The relationship between the OSI network stack and the FIPA ACL protocol stack

FIPA Interaction Protocols	*Request, Query*
FIPA Communicative Acts	*request, query-ref, refuse, agree, failure, inform-done, inform-result, not-understood*
FIPA Messaging	ACL
FIPA Encoding	Bit-Efficient
Transport	ISO-TP
Network	
Data Link	CAN 2.0
Physical	

Fig. 5 The HARP middleware communication stack, which supports both ACL and real-time messaging

extensive specifications ranging from transport protocols and message encoding to agent interaction protocols. However, FIPA does not concern itself with the lower levels of the Open Systems Interconnection (OSI) network stack. The FIPA communications model is a service oriented model, and can therefore be described as its own protocol stack residing within the application layer of the OSI network stack [32]. This relationship is shown in Fig. 4.

The HARP middleware ACC is not FIPA compliant, as its message transfer service does not implement a FIPA message transfer protocol. However, it does implement the higher levels of the FIPA communication stack, including the encoding, messaging, communicative acts and interaction protocols. An overview of the HARP middleware communication stack can be seen in Fig. 5. Its design allows for FIPA ACL messages, as well as real-time packets, to be sent between agents.

The physical and data link layers are implemented using CAN 2.0A. CAN is a multi-master serial bus used extensively in the automobile industry and other real-time environments. A CAN frame is composed of an 11-bit identifier (ID), up to eight bytes of payload, a 15-bit CRC code and several other flow control bits. Arbitration is performed based on the frame ID, requiring each node on the network to use unique IDs. A message with a lower numerical ID will receive priority on the bus, allowing deterministic network timing. Arbitration, message framing, acknowledgement and error detection are all taken care of by physical CAN controllers which interface to physical transceivers. CAN receivers can selectively receive frames based on their message IDs. Thus, a CAN network can be set up to use the frame ID as a destination address, or IDs can be used to indicate message contents.

In the HARP communication stack, CAN IDs are used as message destination addresses. The 11-bit address field is split into a 4-bit cell address (MSBs) and 7-bit protein address (LSBs). Once an agent registers with an AMS, it is given its address ID based on its physical execution location. Protein address 127 is reserved for communicating with the agency services.

Every cell has an internal CAN bus, which links its proteins together. In addition, there is an external, inter-cellular CAN bus linking all cells of a system. These buses are kept separate so that troublesome proteins can be isolated from the rest of the system. To facilitate inter-cellular communication, the cell agency provides a routing service as part of its ACC. The router is responsible for routing CAN frames between the internal and external CAN buses. It does so by reading the CAN frame ID. The router also has the potential to block messages originating from a protein which has been deemed to be troublesome.

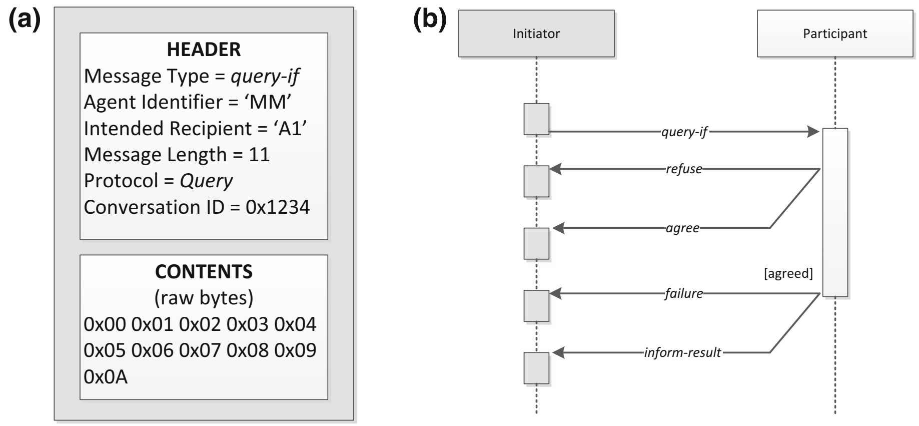
To overcome the eight-byte payload limitation of CAN frames, the ISO 15765-2, or ISO-TP, protocol is implemented. ISO-TP implements the network and transport layers of the OSI network stack. ISO-TP makes use of four frame types: single frame, first frame, consecutive frame, and flow control frame. The transfer of a message begins with the sender sending a first frame, which contains the total length of the message and the first five bytes of message data. The sender then waits for the receiver to respond with a flow control frame, which specifies parameters for the transmission of further frames. Finally, the sender completes the transaction by sending the rest of the message in a series of consecutive frames, each containing seven bytes of message data.

While ISO-TP solves the problem of message size, in its standard form, it is not suitable as an ACL transport service as packets do not contain sender information. Thus, the ISO-TP protocol was modified slightly by adding the CAN-ID of the sender to first frames. This reduces the message content of first frames from six to four bytes. However, it does not affect consecutive frames.

The ISO-TP protocol is used to transport FIPA ACL messages encoded in bit-efficient format. Every message is required to contain a header with several parameters, and message content. The HARP ACL message format is shown in Fig. 6a. It leaves out several of the optional parameters specified by FIPA as these are fixed within the HARP framework. FIPA specifies several encodings for these ACL messages, including XML and String. However, these encodings have large overheads, making them unsuitable for the relatively low-bandwidth CAN buses. Therefore, the FIPA bit-efficient encoding was chosen instead.

Two important parameters of the header are the Communicative Act and the Interaction Protocol. Communicative Acts are essentially message types, as defined by FIPA. These message types are used in interaction sequences known as interaction protocols. An example of an interaction protocol is shown in Fig. 6b. The only mandatory communicative act defined by FIPA is the not-understood message type. In addition to this, the HARP communication stack implements all the communicative acts required for the *Query* and *Request* interaction protocols, as listed in Fig. 5. Examples of interactions include an agent querying the AMS for a registered agent's address, or an agent requesting another agent to start producing real-time telemetry.

The message content will be application specific. However, FIPA suggests implementing a content language with well-defined syntax and semantics to aid the interaction of heterogeneous agents. Examples of content languages include FIPA Semantic Language (SL), Knowledge Interchange Format (KIF) and Constraint Choice Language (CCL). Cur-


(a)

HEADER
Message Type = *query-if*
Agent Identifier = ‘MM’
Intended Recipient = ‘A1’
Message Length = 11
Protocol = *Query*
Conversation ID = 0x1234

CONTENTS
(raw bytes)
0x00 0x01 0x02 0x03 0x04
0x05 0x06 0x07 0x08 0x09
0x0A

Fig. 6 An example HARP ACL message (**a**), and a schematic representation of the Query interaction protocol (**b**)

rent HARP agents have been relatively simple and have operated without a content language, however, the implementation of a content language is planned for the future to enable the creation of more capable and intelligent agents.

3.5 Real-time messaging

Since ISO-TP messages are composed of a number of frames which need to be reassembled at the receiving end, they are not well suited for real-time messaging. Thus, the HARP communication stack also makes provision for raw, real-time CAN traffic. CAN IDs 0-126 are reserved for real-time traffic. Since these IDs have the lowest values, they will always receive priority over ACL messages on the network. The routers on each cell automatically forward all real-time traffic between the internal and external buses, ensuring that real-time messages are distributed as if all proteins were on a common bus. It is envisioned that FIPA ACL messaging will be used by high-level, non-real-time agents for transactions relating to planning and autonomous decision making, while real-time agents will make use of ACL messaging for setting up producer/consumer relations, before switching to raw CAN frames for real-time data transfer.

3.6 Task allocation entity (TAE)

A system can be defined by the list of tasks it is required to be executing at any one time. In systems based on the Satellite Stem Cell Architecture, it is the responsibility of the individual cell agencies to cooperate to ensure that this list of tasks is performed. This involves gathering information about the current system state and potentially creating, relocating, or retiring agents. This decentralised task allocation process is a central part of the HARP middleware.

In biological multicellular organisms, cells secrete chemical messages indicating their current activities into the inter-cellular space. A neighbouring cell can measure the levels of these chemicals in its surroundings to gain information about the current system state. This information influences which proteins get synthesised by the cell’s macromolecular machinery and in effect determines the cell’s own activities.

A simplified version of this process underlies the peer-to-peer (P2P) task allocation strategy employed by the HARP middleware cell agencies. Within each cell agency, a Task

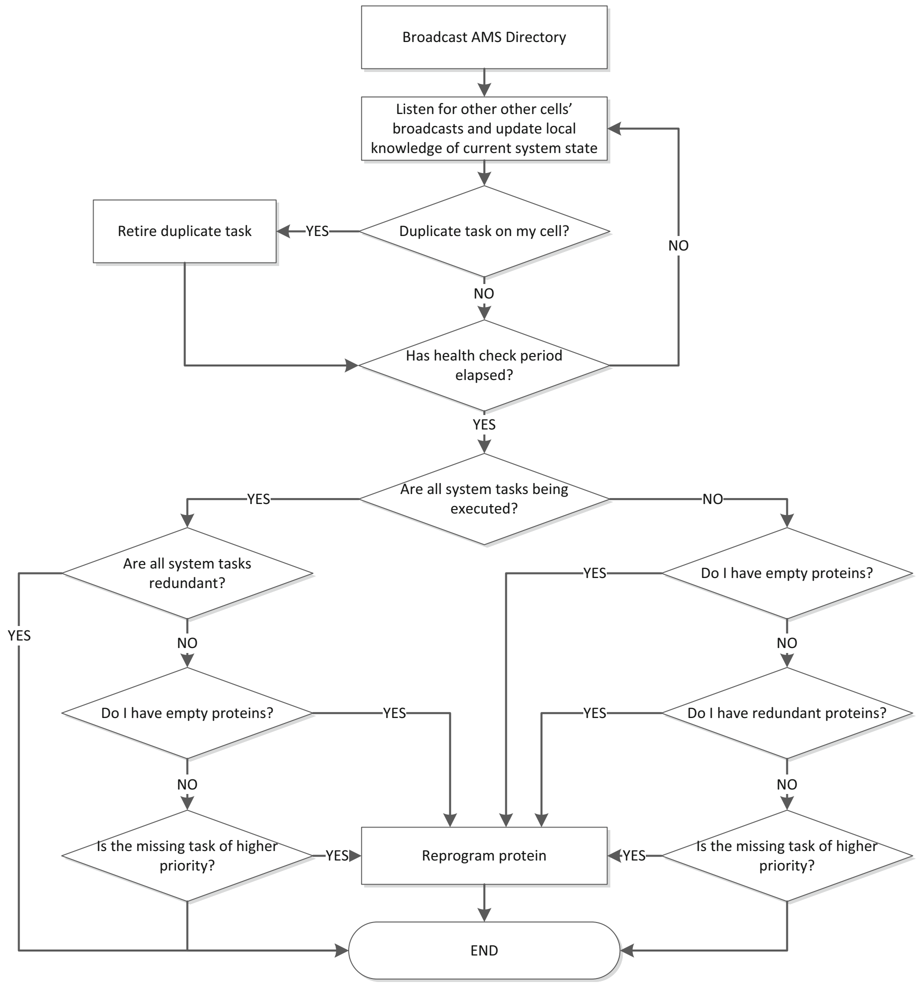

Broadcast AMS Directory
Listen for other other cells' broadcasts and update local knowledge of current system state
Retire duplicate task
YES
Duplicate task on my cell?
NO
NO
Has health check period elapsed?
YES
Are all system tasks being executed?
YES
NO
Are all system tasks redundant?
YES
NO
Do I have empty proteins?
YES
NO
Is the missing task of higher priority?
YES
Reprogram protein
Do I have empty proteins?
YES
NO
Do I have redundant proteins?
YES
NO
Is the missing task of higher priority?
YES
END

Fig. 7 The task allocation strategy employed concurrently and asynchronously by every cell agency's TAE

Allocation Entity (TAE) has the responsibility of performing local task allocation, as summarised in the flowchart of Fig. 7. The TAE queries the AMS and broadcasts the list of registered agents at a user defined interval. By listening out for these messages, every TAE can maintain a list of all agents, and therefore tasks, currently executing somewhere in the system. In addition, every artificial cell contains an identical prioritised list of desired system tasks in non-volatile memory. This list is created by the system developers. The TAE routinely compares the current system state to the desired task list and will request the AMS to attempt to create a new agent if it discovers that a desired task is not currently being executed. Because this process is happening concurrently on all the cells of the system, multiple cells may detect a missing task almost simultaneously. To avoid having the same agents registered on multiple cells, each cell is given a unique priority. The TAE will immediately ask the AMS to retire an agent if it receives information that the agent is also registered on a higher priority cell.

If a cell agency's TAE discovers that a high priority task is not currently executing, but has no free proteins on which to deploy the relevant agent, it will request the AMS to replace one of its lower priority agents with the high priority one. However, this only occurs after a set waiting period during which other cells with free proteins have a chance to discover the missing high priority task. This process is not globally optimum, as the globally lowest task is not necessarily the one which is replaced. However, the replacement will trigger a series of replacements throughout the system, ending with the highest possible set of tasks being executed. This process is not very efficient, but it is robust as no negotiation is required between cells. In addition, agent replacements are expected to occur infrequently and only in response to failures or significant changes in system state (e.g. low power levels).

The allocation of redundancy mentioned in the flowchart is described in Sect. 3.8.

3.7 Health checks and agent mobility

Once an agent has registered with its AMS, its health is continuously monitored at a user definable interval. This health check involves measuring the current consumption of the agent's processor and checking its responsiveness. If an agent fails its health check, the AMS will attempt to revive it. This process involves several steps and may involve processor resets or reprogramming. If it is determined that an agent's processor has suffered unrecoverable damage, the agent can be restored on a different protein processor.

Agent restoration and agent migration follow the same process. While executing, agents may request the AMS to store certain runtime variables in the cell's common memory. This memory is only accessible through the AMS and is analogous to the intra-cellular environment in biological cells. Which runtime variables are stored, and how often they are stored, are up to the agent developer. In addition, every stored runtime variable has an expiration time, after which it is no longer assumed to be valid. This expiration time is also user definable as some runtime variables, such as satellite attitude information, may have a shorter expiration time than others.

To restore an agent, the AMS reprograms an available local protein with the relevant code from central non-volatile memory. Alternatively, the AMS may request a different cell agency to create the agent. Thereafter, it is up to the newly created agent to query the AMS about potential stored runtime variables. This is a soft version of agent mobility as register contents, the stack, and instruction counter cannot be restored.

3.8 Redundancy

In order to hasten the recovery of the system after a failure, redundant agents may be introduced. Two forms of redundant agents exist. Cold redundant agents are proteins which have been programmed with the relevant agent code, but are physically switched off. Hot redundant agents are switched on but in a paused state. Either form can be activated in less time than it takes to reprogram a new protein. If enabled, the P2P task allocation strategy will attempt to fill unused proteins with redundant agents, again starting from the highest priority. Redundant agents are given a lower priority than active agents, ensuring that they are replaced first if required.

3.9 Agency services failure

While failed proteins can be detected and repaired or replaced by the services executing on the MM protein, failure of the MM protein itself is also a possibility. In this case, all agency services, including message routing, will be suspended. As explained in Sect. 2, all the proteins on a cell are electrically identical, allowing any one of them to perform the role of macromolecular machinery. Thus, upon failure of the current MM protein, one of the other proteins should take over its role. However, agent proteins do not contain the software functionality required to reprogram themselves or each other. Instead, upon sensing that an AMS health check has not occurred within a set period of time, an agent protein will enter bootloader mode. Before it does, it will check that no other protein on the cell did so first. In addition, a watchdog timer will timeout, activating a bridge between the internal and external CAN buses. In this state, the remaining proteins can continue to communicate with the rest of the system (in absence of the router) as they are now sitting directly on the inter-cellular bus.

The bootloader will be detected by another cell-agency in the system, as they all continuously monitor the inter-cellular bus. At that point, the saviour cell-agency will reprogram the bootloader-state protein into a new MM protein in a manner similar to programming agent proteins. Once complete, the new MM protein will boot, disable the CAN bridge, and proceed to begin offering the agency services. In addition, it will attempt to revive the old MM protein for reuse as an agent protein. This process can loosely be compared to biological cell division, as one cell gives life to another.

4 Implementation and benchmarking

To determine the practical feasibility of the Satellite Stem Cell Architecture and HARP middleware, a prototype multicellular system was implemented. This section gives a brief description of the prototype cellular hardware, middleware implementation details, and describes the results of several benchmarking experiments that were performed.

4.1 Hardware platform

The hardware platform, seen in Fig. 8, is composed of two artificial cells based on the Satellite Stem Cell Architecture described in Section 2. The hardware was loosely designed on the CubeSat standard [34], in case the opportunity arose to test it on-orbit aboard a Surrey Space Centre CubeSat. Each cell is composed of four artificial proteins, shown schematically in Fig. 9. Each protein is based on an LPC11C24 ARM Cortex M0 MCU, clocked at 48 MHz, with 32 kB of program flash and 8 kB of RAM. These give each testbench cell a processing capacity of 120.96 DMIPS and a power consumption of 390 mW. In addition to the MCU, each protein contains physical interfaces to two CAN buses, an I2C bus, generic I/O circuitry for interfacing to external peripherals, and a discrete I2C node. The MM protein exerts a level of control over the other proteins using these I2C nodes, which physically control the power and bootloader status of their respective proteins.

Each protein MCU contains a CAN bootloader in ROM, which is used by the MM protein during reprogramming sessions. The CAN bus baudrate is limited by the ROM bootloader of the LPC11C24 protein MCUs. Therefore, although bit rates of up to 1 Mbit/s are possible on CAN buses, the internal and external CAN buses of the multicellular hardware platform are operated at an industry standard of 100 kbit/s.

In addition to the four proteins, each cell also contains 256 kB of DNA EEPROM memory for storing eight sets of 32 kB protein firmware, 64 kB of FRAM for runtime variable storage, a discrete watchdog IC and a CAN bus bridge.

A more comprehensive description of the hardware can be found in [33].

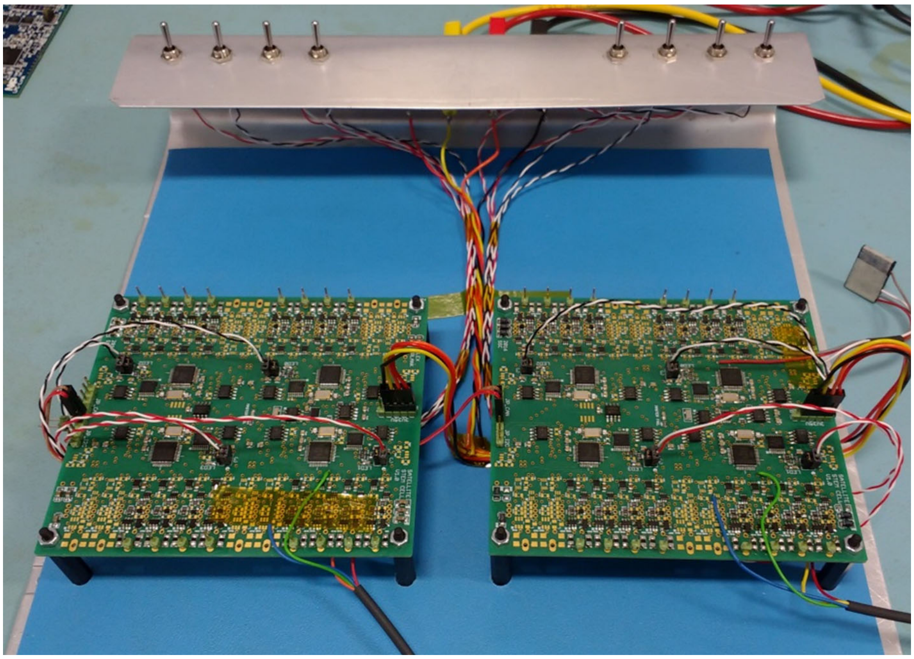


Fig. 8 The hardware testbed for the HARP middleware, composed of two artificial cells based on the Satellite Stem Cell Architecture. Previously published in [33]

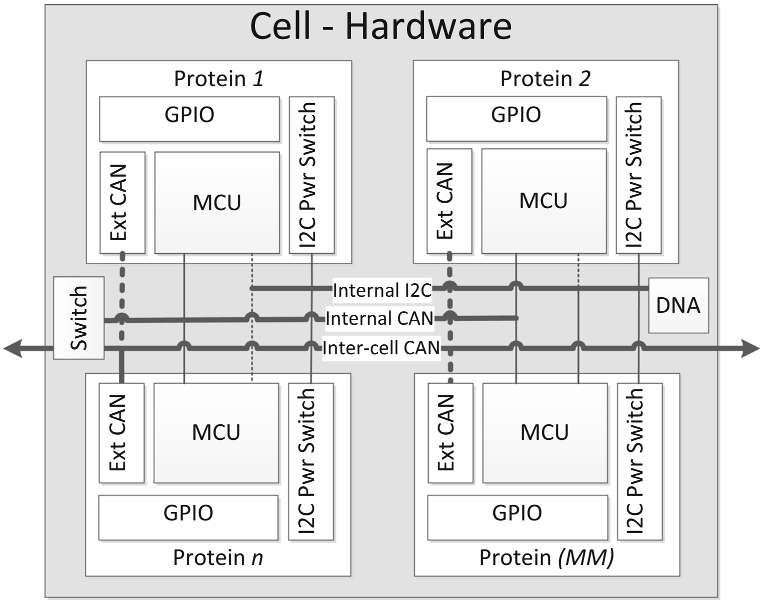


Fig. 9 Each of the four proteins of each implemented artificial stem cell is composed of an ARM Cortex M0 MCU, physical interfaces to two CAN buses, GPIO circuitry, and a discrete I2C power switch

Table 1 The priorities, responsibilities and stack sizes of the FreeRTOS threads composing the HARP agency middleware

Thread	Priority	Implements	Stack size (Bytes)
1	lowest	Health check	1792
2	medium	AMS	1712
3	medium	Kick watchdog Broadcast agent list	624
4	high	Router	544

4.2 Middleware implementation details

An implementation of the HARP middleware was developed for the prototype cell hardware. It is built on top of FreeRTOS, which is a free, open-source, real-time operating system for embedded environments. FreeRTOS has been used on board several low-cost satellites and ports are available for many architectures [35,36]. The middleware is composed of two parts: the agency services and an agent template. The agency services are implemented as a set of four FreeRTOS threads (excluding a default idle thread) and compiled as a single set of firmware for execution on the MM protein. The duties of individual threads, together with their required stack sizes, are shown in Table 1.

FreeRTOS V7.1.0 for the LPC11C24 MCU was used, together with the LPCXpresso development environment. The MM firmware, composed of FreeRTOS, combined with the implemented agency services and relevant hardware drivers, compiles to 28.7 kB with compiler size optimisation enabled. This is approximately 90% of the LPC11C24's flash memory.

The agent template is also based on FreeRTOS and contains a FreeRTOS thread for handling ACL communication, a thread for monitoring the health of the MM (see Sect. 3.9), and a library for utilising the attached generic I/O hardware. The agent template compiles to 24.9 kB, or 78%, of an LPC11C24's flash memory. While the remaining space available for user code is limited, the provided threads and libraries allow user code to focus on high-level agent functionality. Alternatively, agents need not be based on the template, or even FreeRTOS, as long as they are capable of registering with the AMS and responding to ACL-based health checks. However, such agents are not recommended as they may not be capable of detecting and replacing a failed MM protein.

A comparison of the memory requirements of the agent platforms JADE-LEAP, Mobile-C, HARP, and Agilla are shown in Table 2. As these numbers were generated under different conditions, they can only be used as a rough, order-of-magnitude comparison. However, it is evident that HARP's memory footprint is orders of magnitude smaller than traditional agent environments designed for embedded applications.

Due to RAM constraints of the platform hardware (8 kB per protein), some limitations have been placed on ACL messages. Firstly, agent names, which are represented as character arrays, are limited to two bytes in length. Secondly, message contents are currently restricted to 11 bytes. Both of these limitations could easily be removed if the middleware were ported to a system with more RAM.

4.3 Demonstrating functionality

A set of experiments were carried out on the multicellular testbed to demonstrate the full functionality of the HARP middleware. Through the use of toggle switches, permanent and

Table 2 Order of magnitude comparison of the memory requirements of agent platforms designed for embedded applications [26,27]

Agent platform	ROM requirement (kB)	RAM requirement kB)
JADE-LEAP	17782	600
Mobile-C (emb. Ch Interpreter)	~ 3000	
HARP	28.7	8
Agilla	57	3.3

temporary failures of proteins could be introduced to the system. Additionally, multicolour LEDs, added as peripherals via the generic I/O circuitry of all proteins, served as a convenient indicator of agent activity.

The first experiment exercised the decentralised task allocation process. The non-volatile DNA memory of each cell was loaded with the firmware for the MM and three unique agents. The experiment began with only an MM executing on each cell. After approximately 20 seconds, the three agents had been deployed onto the proteins by the MMs via the MCU reprogramming process. Thereafter, the MMs proceeded to fill the remaining proteins with cold redundant versions of the three agents.

The task allocation strategy was tested by severing the inter-cellular link between the two cells. This caused each cell to believe it was the only one in the system, and they each proceeded to deploy all three agents. Once the inter-cellular link was re-established, the duplicate agents were demoted to cold spares. Further testing involved removing power to individual proteins, simulating failures. The loss of an agent due to protein failure resulted in the agent reappearing on a different protein, sometimes on a different cell. If no spare proteins were available and the lost agent was of high priority, it was seen to take the place of a lower priority protein. Such an event has the potential to cause a cascade of reprogramming events, but the system always settled into a state which includes the highest priority agents. This demonstrates the ability for graceful degradation, as the highest priority agents are always kept running, while lower priority agents may be forced to stop as failures build up.

An improved recovery time was noted for agents which had cold-redundant counterparts, with cold redundant protein recovery taking less than a second, while reprogramming a protein takes around 10 seconds.

The reaction of the system to the loss of an MM protein (caused by the removal of its power) was also tested. After the cell watchdog timed out after a few seconds, the CAN bridge was enabled and one of the remaining functional proteins ceased executing its agent and entered bootloader mode. The MM on the second cell successfully noticed the appearance of a bootloader-state protein on the inter-cellular bus and reprogrammed it into a new MM. Once the new MM booted, the CAN bridge was disabled and the system settled back into a steady state containing all three agents.

A second experiment aimed to test the persistence of agent runtime variables. In this experiment, only a single agent was employed, which flashed its LED in a predetermined sequence of colours. After every colour transition, the agent used the Request interaction protocol to ask the MM to store the current state of its colour sequence in central non-volatile memory. When the agent was interrupted by an induced failure of its protein, the agent would reappear on a different protein and continue the sequence from where it left off. To continue the sequence, upon power up and after registering with its AMS, the agent would use a Query

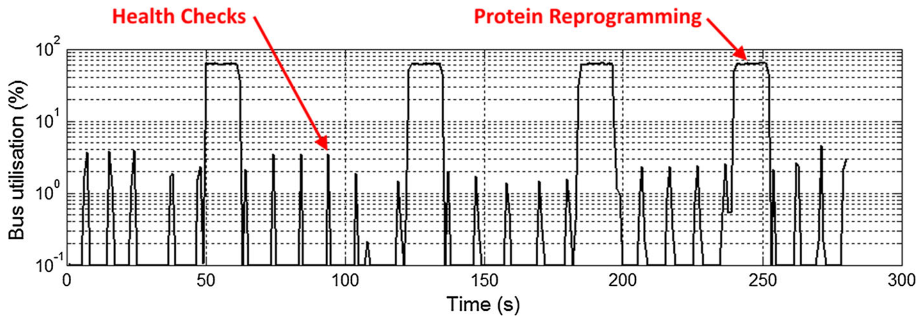


Fig. 10 Internal CAN bus traffic showing bandwidth utilisation during routine health checks and reprogramming events. Previously published in [33]

interaction to ask the MM for any relevant saved runtime variables. This process was tested several times to ensure it worked across cells, too.

To test the expiry of runtime variables, all proteins on both cells were deactived, with the exception of the MMs and the one agent. In this configuration, when the agent's protein was deactivated, too, there were no available proteins onto which the agent could be redeployed by the system. The system was left in this state until the LED-colour runtime variable should have expired (10 seconds in this case). At this point a random protein was reactivated and the agent was seen to restart on it. As expected, the agent had to start from the beginning of its colour sequence.

4.4 Performance benchmarking

Analysing the performance of the testbed Satellite Stem Cell hardware and HARP middleware is important for determining its overheads and scaling potential.

Despite the bit-efficient encoding of ACL messages, ACL communication is expensive in terms of bandwidth. A typical ACL message is 70 bytes long and interaction protocols require the exchange of at least two messages. Thus, communicating an 11-byte payload from one agent to another results in at least 140 bytes of transfer layer traffic. At the ISO-TP layer, this will translate to 22 CAN frames (1 start frame, 1 flow control frame, and 20 consecutive frames). Finally, 22 CAN frames require 2376 bits (297 bytes) of physical link traffic.

Figure 10 shows the internal CAN bus traffic overhead caused by the MM protein's routine health checks, and by the protein reprogramming sequence. Each health check consists of a single ACL query transaction per protein, and therefore creates approximately 10 kilobits (2376 x 4 proteins) of CAN traffic. Since the CAN bus has a bandwidth of 100 kbits/s, the theoretical maximum health check rate is 10 Hz. However, in practice a much slower health check rate should be utilised to avoid overloading the CAN bus. During the experiment shown in Fig. 10, a health check rate of 0.1 Hz was implemented, leading to an average bandwidth overhead of 1%.

Recovery time after failure is dependent on several variables, including the failure type, health check rate, current system redundancy levels, agent size, task priority and number of stored runtime variables (if it was an agent protein failure). However, given knowledge of these variables, the recovery time is deterministic. Practical recovery times were investigated by inducing failures in the prototype multicellular system. Several such failures and subsequent recoveries can be seen in Fig. 10. Protein failures were introduced at approximately 40 and 110 seconds, causing the large peaks in bus utilisation as reprogramming occurred.

Reprogramming takes approximately 10 seconds for a 32 kB agent. At approximately 180 and 240 seconds, the failed proteins were reactivated, leading to the system reprogramming them into redundant roles. The reprogramming routine can consume up to 65% of the CAN bus bandwidth. However, it is important to note that real-time CAN bus traffic always receives priority.

On the prototype multicellular system, with 0.1 Hz health check cycle, recovery times are on the order of 30 seconds. Thus, the system is not appropriate for situations which require fail-operational capabilities. However, for a satellite in orbit such a recovery time is generally tolerable, especially if the system negates the need for a ground controller to get involved in the recovery process. Faster recovery times would be achievable given more internal bus bandwidth.

In addition to bus bandwidth, the other limiting factor in system performance is the processing capacity of proteins. The processing capacity of the ARM Cortex M0 MCU at the heart of the MM protein limits the maximum number of serviceable AMS transactions to 20 per second. In addition, the maximum average bandwidth of the router service is 500 CAN frames per second. Exceeding these specifications starves the health checking routine of processing time. Thus, the maximum practical number of proteins per cell is limited by the MM protein's processing capacity.

5 Case study

To clarify the envisioned usage of the Artificial Stem Cell architecture and HARP middleware, a case study based on 'cellularising' a typical microsatellite's avionics suite is presented. In contrast to CubeSats, which fall towards the lower end of the small, low-cost satellite scale, microsatellites, with a mass of 10 - 100 kg, represent the most commercially viable (to date) class of small, low-cost satellites.

Table 3 lists a representative set of microsatellite avionics which could feasibly be replaced by the current generation of testbed artificial cells. These subsystems are based on those of a typical Surrey Satellite Technology Ltd microsatellite bus, which has a cross-strapped, dual-string, subsystem-based architecture [37]. Certain subsystems, such as the electrical power system and communication subsystems, currently contain too much custom analogue circuitry to be feasibly replaced. However, future generations of artificial cells, through the advancement of technologies such as software defined radio, are predicted to be even more multifunctional.

With the exception of the onboard computer, OBC750, all the other subsystems are essentially peripheral controllers. They are based on 8-bit MCUs, whose processing capacity can easily be matched by a single testbed protein. Therefore, cellularising these subsystems is simply a matter of ensuring that sufficient generic I/O channels exist. The testbed proteins each have 6 six I/O channels, resulting in the required number of proteins, per subsystem, given in Table 3. The tasks performed by these subsystems are all required to be executed in real time.

The SSTL onboard computer, OBC750, requires 10 proteins to match its 500 DMIPS processing power [38]. Its workload is composed of several tasks, allowing a natural distribution of the tasks to several agents executing on discrete protein processors. The OBC750 performs a mixture of real-time and non-real-time tasks. Real-time tasks include attitude estimation, orbit propagation and attitude control. The agents executing these tasks will use ACL messages to set up publish-subscribe relationships with the real-time agents controlling the peripherals. Once the relationship is established, the transfer of data will be based on

Table 3 Representative microsatellite subsystems, their processing capacity, associated peripherals, and equivalent number of proteins

Subsystem	Redundant	DMIPS	Peripherals	Eq. proteins (per sub.)
OBC750	Cold	500	none	10
AIM ADCS interface	Cold	< 10	$3 \times$ magnetorquers	3
			$1 \times$ magnetometer	
			$1 \times$ fine sun sensor	
SP10 Reaction wheel	3-out-of-4	< 10	$1 \times$ BLDC Motor	1
Propulsion subsystem	Cold	< 10	$1 \times$ heater	1
			$2 \times$ valves	
Gyroscope interface	Cold	< 10	$1 \times$ digital gryoscope	1

real-time CAN frames. For example, the attitude estimation agent will request the sun sensor agent to periodically transmit its sensor reading as a real-time CAN frame, and may occasionally request a certain calibration value or filtering coefficient to be changed. Similarly, the attitude control agent will use ACL-based interactions to set up a periodic broadcast of attitude data from the attitude estimation agent.

In parallel to the real-time agents, other agents will be performing non-real-time tasks. Examples include payload operations and planning, environmental monitoring, and routine housekeeping operations. Such agents will interact largely through ACL messages, as they do not handle large quantities of data or have real-time requirements. Such agents are important as satellites are typically out of ground station communication range for much of their lifetimes. While the use of intelligent agents on board satellites has been limited to date, there is a lot of potential for agents to automate tasks, especially if the interaction can be extended to ground station agents and agents in other satellites. Examples of useful, non-real-time agents could include an agent which plans and executes a rendezvous with another satellite, or an agent which plans payload operations based on weather forecasts.

While the HARP communication stack transport layer does not implement a FIPA standard, the satellite's radio transceiver, or more likely the ground station, can act as a gateway. Since the HARP protocol stack is FIPA compatible from the bitwise-encoding layer upwards, a gateway would be simple to implement. In this way, ground station agents, based on more traditional agent platforms, could interact with the HARP agents operating on the satellite. To truly harness the power of autonomous agents, a semantic language must be added to the current HARP protocol stack. This is planned for the future, but will likely require a new generation of artificial cells, based on MCUs with more memory.

Based on Table 3, a multicellular avionics suite composed of 10 testbed cells is envisioned. Such a system would provide 1200 DMIPS of processing capacity at a power consumption of 4W. In comparison, the OBC750 itself consumes 10W. This power saving is largely due to the use of modern, low power MCUs in the cell-based design. Furthermore, the cell-based design has the ability to linearly scale its power consumption with processing capacity by physically shutting off proteins.

6 Conclusion

This paper described the design and implementation of the Hybrid Agent Real-Time Platform (HARP), which is designed to take advantage of the unique, distributed hardware of the Satellite Stem Cell Architecture. Together, the HARP middleware and Satellite Stem Cell Architecture aim to improve the reliability and capabilities of small, low-cost satellites by decreasing manufacturing and integration costs and enabling responsive, on-orbit reconfiguration and graceful degradation.

The HARP middleware follows the Foundation for Intelligent Physical Systems Abstract Architecture, by implementing an Agent Management System, Agent Communication Channel, and Agent Execution Environment. In addition, the HARP communication stack follows the FIPA specifications for ACL messages, encoding, commutative actions and interaction protocols.

Unlike the majority of agent platforms which run on desktop machines, the HARP middleware is expected to run on the resource constrained, embedded environment of a low-cost satellite. This environment necessitates a very small memory footprint, fault tolerance, and support for real-time operations. To fulfil these requirements, the HARP middleware distinguishes itself from typical agent platforms in three areas. Firstly, the HARP middleware uses discrete processors, instead of virtual machines or interpreters, as its agent execution environment. This has the advantage of reducing the agency memory footprint and enabling agents to perform real-time tasks. Secondly, the HARP communication stack makes use of ISO-TP over CAN 2.0A as its transfer level protocol, cutting out resource-intensive layers such as HTTP and IIOP. In addition, the communication stack allows real-time CAN traffic to share the network and be given priority over ACL communication. Finally, the HARP middleware embeds a peer-to-peer task manager in each agency, allowing systems which are built using the Satellite Stem Cell Architecture and HARP middleware to autonomously reconfigure in the event of failures. Agencies monitor the health of their agents, and agents can monitor the health of their agencies. In the event of a failure, both agents and agencies have the ability to be moved from one processor to another. In addition, redundant agents can be proactively deployed to reduce reconfiguration time.

The HARP middleware was implemented on top of the free, real-time operating system FreeRTOS, and deployed onto a prototype multicellular system composed of two artificial cells of four proteins each. The middleware consists of two parts, a set of agency services which run on MM proteins and an agent template. The services firmware has a memory footprint of just 28.7 kB and requires less than 8 kB of RAM, largely due to the offloading of the agent execution environment to dedicated processors. The agent template, which includes functions for ACL communication and generic I/O control, occupies just 24.9 kB of memory.

A series of experiments were successfully performed on the prototype multicellular platform to exercise the full functionality of the HARP middleware. Additionally, through a set of benchmarking experiments, the scaling potential of the system was investigated. In addition to the 11-bit CAN addressing scheme limiting the system to a maximum of 15 cells and 126 proteins per cell, various limits are imposed by the current hardware. Firstly, the overhead of ACL communication and the limited bandwidth of the internal CAN bus limit practical health check rates to < 1 Hz. With a health-check rate of 0.1 Hz, recovery times on the order of 30 seconds were observed. Secondly, the MM protein MCU's processing capacity limits the maximum number of AMS transactions per cell to 20 per second. Finally, each cell's message routing service has a maximum average bandwidth of 500 CAN frames per second.

Despite these limitations, the HARP middleware and Satellite Stem Cell Architecture form a practical and feasible alternative to current low-cost satellite architectures. The initial development of the proposed architecture is more complex than that of a system based on unique, discrete subsystems. However, once the artificial cell hardware is in production and the middleware has been fully tested, the development of future systems becomes streamlined. This process should simply involve determining the required number of cells, based on processing capacity, I/O, and redundancy requirements, and developing a set of template-based agents to perform system tasks. Planned future work includes the addition of a semantic language to the HARP protocol stack, the development of ground station software based on a more traditional agent platform, and research into more advanced task allocation strategies, such as optimising for bus bandwidth or reconfiguration time.

References

1. Erlank, A. O., & Bridges, C.P. (2015). A multicellular architecture towards low-cost satellite reliability. In *2015 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, pp. 1–8.
2. Erlank, A. O., & Bridges, C. P. (2016) The satellite stem cell architecture. In *2016 IEEE Symposium Series on Computational Intelligence (SSCI)*, pp. 1–8.
3. Bridges, C. P. (2009). *Agent Computing Platform for Distributed Satellite Systems*. Ph.d.: University of Surrey.
4. McKinnon, A. D., Dorow, K. E., Damania, T. R., Haugan, O., Lawrence, W. E., Bakken, D. E., & Shovic, J. C. (2003). A configurable middleware framework with multiple quality of service properties for small embedded systems. In *Proceedings—2nd IEEE International Symposium on Network Computing and Applications, NCA 2003*, pp. 197–204.
5. Fowell, S. D. (2003). Launching real-time corba into space: Motivation and implementation. In *OMG's Workshop on Distributed Object Computingfor Realtime and Embedded Systems*. VA: Arlington.
6. Gokhale, A., & Schmidt, D. (1999). Techniques for optimizing CORBA middleware for distributed embedded systems. In *IEEE INFOCOM '99. Conference on Computer Communications. Proceedings. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. The Future is Now (Cat. No.99CH36320)*, pp. 513–521 vol. 2, IEEE.
7. Henning, M. (2006). The rise and fall of CORBA. *Magazine Queue—Component Technologies*, *4*(5), 28–34.
8. Hadim, S., & Mohamed, N. (2006). Middleware: Middleware challenges and approaches for wireless sensor networks. *IEEE Distributed Systems Online*, *7*(3), 1–23.
9. Levis, P., & Culler, D. (2002). Mate: A tiny virtual machine for sensor networks. In: *Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 85–95.
10. Madden, S. R., Franklin, M. J., Hellerstein, J. M., & Hong, W. (2005). TinyDB: an acquisitional query processing system for sensor networks. *ACM Transactions on Database Systems*, *30*(1), 122–173.
11. Razzaque, M. A., Milojevic-Jevric, M., Palade, A., & Clarke, S. (2016). Middleware for internet of things: A survey. *IEEE Internet of Things Journal*, *3*, 70–95.
12. FIPA. (2016). *Welcome to the foundation for intelligent physical agents*. Accessed June 9 2016.
13. Kravari, K., & Bassiliades, N. (2015). A survey of agent platforms. *Journal of Artificial Societies and Social Simulation*, *18*, 11.
14. Bellifemine, F., Caire, G., & Greenwood, D. (2007). *Developing multi-agent systems with JADE. Wiley series in agent technology*. Chichester, UK: John Wiley & Sons Ltd.
15. Bellifemine, F., Bergenti, F., Caire, G., & Poggi, A. (2005). *Jade—A java agent development framework* (pp. 125–147). Boston, MA: Springer US.
16. Poslad, S., Buckle, P., Hadingham, R. (2000). The fipa-os agent platform: Open source for open standards. In *5th International conference and exhibition on the practical application of intelligent agents and multi-agents*, p. 335.
17. Krol, D., & Nowakowski, F. (2013). Practical performance aspects of using real-time multi-agent platform in complex systems. In *2013 IEEE International Conference on Systems, Man, and Cybernetics*, pp. 1121–1126.
18. Frantz, C., Nowostawski, M., & Purvis, M. (2010) Multi-agent platforms and asynchronous message passing: Frameworks overview (information science discussion papers series no. 2010/07).

19. Laukkanen, M., Tarkoma, S., & Leinonen, J. (2002). *FIPA-OS Agent Platform for Small-Footprint Devices* (pp. 447–460). Berlin, Heidelberg: Springer.
20. Bergenti, F., & Poggi, A. (2002). Ubiquitous information agents. *International Journal of Cooperative Information Systems*, *11*(03n04), 231–244.
21. Bergenti, F., Poggi, A., Burg, B., & Caire, G. (2001). Deploying fipa-compliant systems on handheld devices. *IEEE Internet Computing*, *5*, 20–25.
22. Bergenti, F., Caire, G., & Gotta, D. (2014). Agents on the move: JADE for android devices. In *Proceedings of the XV Workshop "Dagli Oggetti agli Agenti", Catania, Italy, Sept. 25–26*.
23. Koch, F. L., & Meyer, J.-J. C. (2003). Knowledge-based autonomous agents for pervasive computing using agentlight. *IEEE Distributed Systems Online, 4*(6). https://doi.org/10.1109/MDSO.2003.10002.
24. Oracle, (2017). *Frequently asked questions oracle java me embedded 8 and 8.1*. Accessed February 19.
25. Chen, B., Cheng, H. H., & Palen, J. (2006). Mobile-c: A mobile agent platform for mobile c-c++ agents. *Software: Practice and Experience*, *36*, 1711–1733.
26. Fok, C.-L., Roman, G.-C., & Lu, C. (2009). Agilla: A mobile agent middleware for self-adaptive wireless sensor networks. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, *4*, 16:1–16:26.
27. Bridges, C. P., & Vladimirova, T. (2011). Real-time agent middleware experiments on java-based processors towards distributed satellite systems. INin *2011 Aerospace Conference*, pp. 1–10.
28. Zabel, M., Preuber, T. B., Reichel, P., & Spallek, R. G. (2007). Secure, real-time and multi-threaded general-purpose embedded java microarchitecture. In *10th Euromicro Conference on Digital System Design Architectures, Methods and Tools (DSD 2007)*, pp. 59–62.
29. Bernard, D. E., Dorais, G. A., Fry, C., Gamble, E. B., Kanefsky, B., Kurien, J., Millar, W., Muscettola, N., Nayak, P. P., Pell, B., Rajan, K., Rouquette, N., Smith, B., & Williams, B. C. (1998). Design of the remote agent experiment for spacecraft autonomy. In *Aerospace Conference, 1998 IEEE*, vol. 2, pp. 259–281.
30. Surka, D. M., . Brito, M. C, & Harvey, C. G. (2001). The real-time objectagent software architecture for distributed satellite systems. In *Aerospace Conference, 2001, IEEE Proceedings.*, vol. 6, pp. 2731–2741.
31. Jopling, C., Boue, S., & Belmonte, J. C. I. (2009). Dedifferentiation, transdifferentiation and reprogramming: Three routes to regeneration. *Nature Reviews. Molecular Cell Biology*, *12*, 79–89.
32. Posland, S. (2017). *Review of fipa specifications*. December 2006. Accessed Feb 27.
33. Erlank, A. O., & Bridges, C. P. (2017) Satellite stem cells: The benefits and overheads of reliable, multicellular architectures. In *2017 IEEE Aerospace Conference*.
34. Puig-Suari, J. (2017). Cubesat developer resources. Accessed Feb 19.
35. Bridges, C. P., Kenyon, S., Shaw, P., Simons, E., Visagie, L., Theodorou, T., Yeomans, B., Parsons, J., Lappas, V., & Underwood, C. et al. (2013). A baptism of fire: The strand-1 nanosatellite. In *AIAA/UTU Small Satellite Conference*.
36. Sünter, I. (2014). *Software for the ESTCube-1 command and data handling system*. Ph.D. thesis, Tartu Ülikool.
37. Sweeting, M. (2001). 25 Years of space at SurreyPioneering modern microsatellites. *Acta Astronautica*, *49*(12), 681–691.
38. Surrey Satellite Technology Ltd, "OBC750," 2014.