Protocol For Code Exchange In Survivable Embedded Systems

Michael E. Locasto
Department of Computer Science
Fu Foundation of Engineering and Applied Science
Columbia University
locasto@cs.columbia.edu

Abstract

As the cost of both networking and producing powerful embedded devices drops, collections of these highly specialized and heterogeneous platforms will proliferate. These networks will face security threats and suffer traditional hardware failure. Failure of embedded devices is undesirable because these devices often perform critical functions and are difficult to take offline and upgrade. Networks of embedded devices require a method to accomplish functional survivability of essential computations in a hostile or volatile environment.

This paper presents a protocol and describes an implementation for migrating essential computations from failed devices. An essential computation is a device's primary algorithmic functionality. This migration is accomplished by specifying both an Area Controller (AC), which contains definitions for every essential computation, and a number of proxy agents, which periodically send the AC update statements. The AC and proxy agents negotiate the specifics of process migration when the AC has determined that a device has failed.

The implementation of this protocol in a network of embedded devices is a crucial step toward fault tolerance and survivability in networks of embedded devices. The protocol can also be applied to networks that do not involve embedded systems.

Keywords

Fault tolerance, embedded systems, process migration, network availability, survivability, security protocols

Introduction

Even though the dedicated efforts of hardware and software engineers have enabled computing devices to become more or less reliable appliances, there are many domains where any failure (either malicious or arbitrary) of a computing device is completely unacceptable. In the non-embedded domain, many organizations devote scads of money and countless human-hours to assuring that their web services systems remain in a High-Availability (HA) state. The amount of hardware, software, networking, and management in such systems is staggering, if not overwhelming.

We can observe how much effort is put into serving web pages, and then consider the extreme requirements for networks of embedded devices needed to fly airplanes, perform health monitoring in hospitals, and control nuclear power facilities. As embedded devices are networked together, not only do they face traditional failures, but also the growing legion of threats made possible by a networked environment.

Fault tolerance and survivability of computer systems and networks is often addressed by both replication of critical services (distributed databases, server clusters), and redundancy (UPS, dual network connections). Traditionally, work has been done to guarantee some level of service by the system or network in the presence of attack, failure, or high load [10].

The protocol presented in this paper continues this theme by precisely specifying the steps necessary for ensuring that an essential computation can be migrated to a target device based on some optional policy specification. The result of ensuring that an essential computation may be migrated is the continued execution of critical algorithms in the network.

1. Related Work

The impetus for this work is detailed in Keromytis, et al [2], which describes an ambitious and detailed plan to design programming languages, policy languages and compliance-checking mechanisms, and dynamic update mechanisms to meet the challenges presented by survivability in embedded network environments.

Faults in and failures of computing devices have long been an area of concern for computing professionals. The design of this protocol involves three areas of computer science: embedded systems, process migration and distributed computation, and network survivability.

1.1 Embedded Systems

Designing embedded systems to be robust is a difficult and time consuming process because of the extreme constraints involved in the available hardware environment. Embedded systems have more extreme power, heat, speed, and space requirements than ordinary computer hardware [2]. In addition, Edwards et al [9] note that most design of embedded systems was done on an ad hoc basis as recently as five years ago, with little or no formal specification or proof of correctness. Edwards et al [9] have written a detailed analysis and presentation of formal methods for specifying, validating, and synthesizing reactive real-time embedded systems. Their approach identifies "management of both design complexity and system heterogeneity as the key problem." It is clear that heterogeneity is a necessary challenge in embedded systems, and any work done for embedded systems networks would do well to adopt the approaches detailed in their paper.

1.2 Process Migration

Process migration is a technique that has been studied for some time with the intent of providing load balancing for clusters of servers or workstations. Eager et al [4] had suggested that process migration added too much overhead in general for the anticipated benefits, but Downey and Harchol-Balter [1] refuted this claim and presented several common environments in which process migration is largely beneficial. Process migration is well suited to clusters of computers and is most famously implemented as part of the MOSIX [3,14] distributed operating system software.

The Charlotte system is another environment for process migration research. Charlotte is built with the goal of ensuring that the migration completes successfully. Artsy and Finkel [12] provide both a succinct overview of process migration theory and some performance characteristics related to Charlotte.

The technique of process migration is often compared with remote execution, which is presented as a lower-cost alternative. However, process migration offers true continuity of service, whereas failure of a device providing remote execution services necessitates the availability of a complete replica - something that is not always practical in an embedded environment. Typical remote execution mechanisms are RPC, Java

RMI, and CORBA. These mechanisms rely on a local proxy to call functions on a server over the network, rather than actually moving a process.

Dynamic updates of software is a complex operation. Hicks [7] points to Smith [5] as an excellent review of process migration techniques. Hicks also describes the notion of state transfer as a process of automatically encoding current process information and details the difficulties involved with the checkpointing technique. The primary difficulty in state transfer is twofold: the target may not understand the state format, and some essential state may be hidden by the operating system of the original machine [7].

The initial version of the PCXSES protocol assumes that with a common virtual machine in the embedded network, state translation can be accomplished, albeit at a higher level than a snapshot of the bits in (a hardware-specific) memory. The protocol does not limit what can be considered state; actually determining what state is important and translating it are left to the implementation, although a mechanism for specifying what state is important and how to update it is presented in Section 2.3.

On the other hand, heterogeneous process migration, as addressed in Smith and Hutchinson [8] does not make the assumption that all devices share a common runtime platform. Smith and Hutchinson [8] give a very precise analysis of the requirements for designing a process migration protocol. Future versions of PCXSES will address the issues raised by a heterogeneous network.

1.3 Network Survivability

The threat model for computer networks is significantly different from the threat model for a standalone system. Networks are vulnerable to a wide variety of attacks, and the technical report by Ellison et al [10] provides a very complete overview of classic network survivability. Fault tolerance and network survivability are two areas of network security that attempt to research and respectively address Byzantine [6] and malicious failures in networks.

Zhang et al [13] have commented that "the composition of most networks tends to converge on a single technology [often from the same vendor] at each layer of the network." This trend may boost interoperability; however, Zhang et al [13] have suggested that homogeneity in networks presents vulnerabilities and that survivability may be achieved through greater heterogeneity.

The weakness of this approach is the greater challenge in configuration and the cost of management when adding non-required complexity to the network. However, in the OASES [2] proposal, Keromytis et al argue that the natural heterogeneity of an embedded network provides a high degree of redundancy. This inherent redundancy may provide a platform for survivability. Furthermore, we should observe that embedded networks contain many highly specialized components that already require diverse configuration and specialized management.

Employing process migration in a network of embedded devices is a highly desirable and cost effective method of assuring the survivability of this network. The PCXSES protocol performs this task.

2. The PCXSES Protocol

The PCXSES protocol is a straightforward series of steps that borrows from general principles of network routing protocols [16] and the idea of timeouts as presented in Lamport et al [6]. The protocol has four phases:

1. GOOD MORNING phase (walk in the door)
2. PARTY phase (continuously greet host)
3. RECOVERY phase (host gives keys to sober friend)
4. GOODBYE phase (people leave party)

The underlying idea is that the AC can store definitions (object code) for every process running on the devices, but needs to be alerted to changes in the state of those processes.

Each device joins the network with a GOOD MORNING message. The device then periodically updates the AC with altered state information via a HELLO or STATE message. Finally, if the AC has determined that a device has failed, it will enter the RECOVERY phase with that failed device and a target device. It will send a MORPH (Mobile Object RePlacement Header) message to the target device. The target device MUST NOT implicitly trust any MORPH message. Entering the RECOVERY phase with one device does not require the AC to abandon the PARTY phase or GOOD MORNING phase with other devices.

The presentation of the PCXSES covers three core components: the description of the protocol phases, the description of the message formats, and the description of the state update mechanism. Finally, a sample implementation [15] in Java is discussed for migrating a running game of PacMan to another device.

Figure 1 displays the various protocol phases and the legal messages during each phase. The MORPH messages are only employed if a device has failed. Normal operation consists of a GOOD_MORNING pair, a variable number of HELLO or STATE messages, and a GOOD_BYE pair.

```
[client] -- GOOD_MORNING {params} -->> [AC]
[client] <<-- GOOD_MORNING {params} -- [AC]

 [client]      -- HELLO  |  STATE -->>      [AC]

[client] <<-- MORPH {^, ids, #b/id, state} --[AC]
 [client]     <<-- object definition #1 --     [AC]
 [client]     <<-- object definition #2 --     [AC]
    [client]              <<--  ....  --           [AC]
 [client]     <<-- object definition #n --     [AC]
  [client]     -- MORPH {ack | fail} -->>     [AC]

  [client]       -- GOOD_BYE -->>       [AC]
  [client]     <<-- GOOD_BYE --         [AC]
```

Figure 1: *PCXSES Protocol Sketch.* The client thread running on the device maintains contact with the AC, notifying it of state updates. If the device fails, the AC begins a MORPH conversation with another device to replace the failed device. A device may also gracefully leave the protocol by sending a GOOD_BYE message. Figure 1 depicts the phases involved in the full protocol.

2.1 Protocol Phases

The protocol phases are straightforward: session setup (GOOD_MORNING), session maintenance (HELLO), failover (MORPH) and session teardown (GOOD_BYE).

2.1.1 Good Morning Phase

This phase allows a device to join the federation or network by contacting the AC. Thus, the AC does not have to poll the network and solicit new devices for their connectivity information. The AC maintains a cache of devices specified in a

configuration file, and activates a device once it receives a GOOD_MORNING message.

2.1.2 Party Phase

In the party phase, each device sends notifications to the AC that the device is still alive. The device may also send updated state information as part of the notification or as part of a different message. This phase is critical; should the AC fail to receive some given number of HELLO messages in a given time period, the AC will consider the device dead, act to replace it, and ignore further communications from it.

2.1.3 Recovery Phase

Naturally, the RECOVERY phase is the most interesting. This phase begins with the AC determining a suitable target device for the failed computation according to its policy rules. Once the AC selects a device, the AC retrieves the necessary object code from its repository, and sends these bytes, along with the LKGS and some metadata, to the target device. The AC first sends a Bottle object of type MORPH with the metadata so that the receiving client thread can properly parse the received object code stream. The client thread receiving the new computation then notifies the AC if the transfer has been successful, and attempts to create the new computation, along with using the State Update Interface (SUI) agent to configure the new computation to the LKGS.

2.1.4 Goodbye Phase

This phase allows the protocol to terminate gracefully. If this phase were not included in the protocol, the AC would assume that a device had failed and being a needless Recovery Phase.

2.2 Message Formats

All messages are encapsulated in a BOTTLE object with appropriate type identifiers. There are five types of messages:

1. GOOD_MORNING message
2. HELLO message
3. STATE message
4. MORPH message
5. GOOD_BYE message

All BOTTLEs contain the same fields, some of which may be unused depending on the message type. A BOTTLE contains the message type, the client name, the mode, the hello interval, the dead interval, the state, the object code identifiers, the number of bytes per object, control signals, and flags to indicate which objects should be invoked after a MORPH.

The GOOD_MORNING message is used by the client to notify the AC of its existence and to negotiate parameters (mode, hello interval, and dead interval) to be used in the rest of the protocol. Mode is either SPLIT_MODE or JOINED_MODE. The JOINED_MODE is more common and used to indicate that a HELLO message includes state information. The SPLIT_MODE is used to indicate that STATE messages are sent independently of HELLO messages. A STATE message will not reset the timer that controls the "alive" flag for the device.

The HELLO message is used to notify the AC that the device is still active, and may be used to pass state information. Providing the option of separating state from the HELLO message is a performance enhancement; it also provides the device programmer more control over network traffic. The MORPH message is used to migrate a process to a target device and acknowledge that the migration has successfully completed. Finally, the GOODBYE message offers a graceful mechanism for a device to notify the AC that it has left the network and does not need to be "recovered."

2.3 State Transfer and Update Definition Meta Language

State checkpointing and recovery of the Last Known Good State is critical. For example, software that tracks the flight path of a missile should have the most current information possible. However, transfer of a computation's state is difficult precisely because every computation has a unique state, and the form of this state is alien to any other computation (not to mention the underlying target hardware).

In order to transfer state, the protocol supplies the notion of a State Update Interface (SUI). This construct is based on the fact that most computations have an informal collection of methods, functions, or procedures that represent an "update API," wherein some of the methods represent ways to obtain current state information and other methods represent ways to set the current state information. Each device knows how to handle its state the best. This functionality can be captured in a SUI agent so that the device can "share" this knowledge with the rest of the devices on the network when the need occurs.

The SUI object is transferred from the AC to the target device VM along with the new compution. The SUI is unique to each device and acts as a knowledgeable agent to transfer the LKGS of that device to the new computation running in the target device.

For example, in a game of PacMan, the current state of the game can be represented by the x,y coordinate locations of PacMan and the BadGuys. In addition, lives left, level, and score are also part of the state. Lastly, a flag for each traversable spot on the board can be kept to indicate whether or not the "spot" has been eaten by PacMan. This state information is all that matters, and can be compactly represented in an integer array, without going through the excruciating detail of saving every object's memory image.

3. Results

The protocol proof of concept and development environment was implemented using the Java 2 Standard Edition version 1.4.1 platform. This implementation was tested for a variety of metrics, most notably network bandwidth consumption, CPU utilization, memory consumption, and recovery time.

3.1 Implementation

The implementation of the protocol is centered around three packages: a protocol package (**locasto.pcxses.protocol**) that encapsulates and specifies the core protocol objects, a client package (**locasto.pcxses.client**) that provides a default implementation of the client functionality, and a server package (**locasto.pcxses.ac**) that provides the AreaController functionality.

The application is supported by several other packages distributed with the implementation. These packages encapsulate logging and configuration utilities.

The protocol package specifies the Bottle object format, the base State class, the base StateUpdateInterface interface, and a client API via the PCXClient interface. The client

package provides a PCXClientFactory object and a default PCXClient implementation, including a MorphReceiver and a HelloTask. The server package contains the various pieces of the AreaController, including the PolicyEngine, RepositoryManager, and DeviceManager components.

The AreaController uses two XML configuration files to learn about its environment. The first configuration file contains most implementation-specific details and configuration parameters for the various components. The second configuration file specifies all devices that the AreaController is responsible for. This file also contains the object identifiers for the essential computations in each device, as well as the object identifiers of the specific State and SUI objects for each device. This implementation maintains a file directory hierarchy that houses the object code for each essential computation. Other options for this repository include a relational database or LDAP-style directory.

3.2 General Performance Specifications and Observations

The AreaController was hosted on both a single Pentium III processor at 1.0 GHz Windows 2000 (service pack 3) platform with 384 megabytes of RAM and a dual Xeon at 2.0 GHz machine running RedHat Linux 7.3 with 1024 megabytes of RAM. However, the protocol does not require such firepower to perform in a reasonable manner. Indeed, the memory requirements for the AreaController were on the order of fifteen threads running in eight megabytes, and taking on avergage less than three percent of the CPU during normal operation. Network bandwidth consumption remains low because all Bottle messages except for MORPH messages are small, but increases in proportion to the number of devices on the network and the requirements of their individual rates of contact with the AC.

Test devices were hosted on weaker workstations and some network cluster machines, running a variety of Microsoft Windows and Solaris operating systems. Future testing will be performed on smaller devices and embedded platforms.

3.3 Performance Measurements

The protocol functions as specified. Two devices were represented during testing. The first device was a simple chat-style GUI that recorded its input as state. The second device was a game of PacMan. Both devices were allowed to (and successfully did) load the other device's computation. The following sections detail some of the performance metrics collected during testing. Testing was performed using an unbridged shared local Ethernet medium at 10Mb/s and standard Java TCP Socket connections.

3.3.1 Network Bandwidth Consumption

Transport of a Bottle object involves serialization of that object and then sending the serialized bytes over the wire. Java serialization does not transport an entire class definition; rather, it is a representation of all the instance data member variables (not static class member variables) in the object. The average Bottle object holds an integer-specified message type (32 bits in Java), a variable length String object representing the device's identifier, and the serialized bytes of a variable length State object. Therefore, the number of bytes for the average serialized Bottle are quite small, and are dominated by the size of the information contained in the State object. If the state has not changed since the last HELLO message, the client may opt not to send any State information, as the AC already has the most current LKGS.

There are three scenarios in which network bandwidth can be measured. The first is normal operation of the network, where all devices periodically send HELLO messages. Even if the number of devices is great and their update timer has a short period, bandwidth should not be a bottleneck or cause undue packet delay due to the small size of the data being transferred. The second scenario is one in which some devices have failed, placing a moderate load on the AC and causing the existence of some MORPH messages and attendant object byte streams. Bandwidth should be nearly utilized in this scenario for the duration of the MORPH transactions. However, it should soon stabilize as the AC processes the transactions. The third scenario is the catastrophic loss of many devices, where the AC is placed under heavy load attempting to find surviving devices and satisfy its policy rules. In this scenario, many MORPH messages may be present, but this is not necessarily true in all cases. Only a handful of the numerous failed devices may require a MORPH transport. In addition, the number of MORPH messages is always limited by the number of allowable target devices. In this final scenario, bandwidth may become a bottleneck, especially if the amount of data for state and object definitions to be transferred is large.

In normal operation, with two devices, each running a HelloTask about every 5 seconds, bandwidth utilization is minimal. If one of the devices fails, bandwidth is utilized for an observed maximum of two seconds. Further experimentation with collections of devices at different orders of magnitude will shed light on scalability issues.

3.3.2 Network Errors

There were some observable network socket errors, mostly due to collision of packets on the Ethernet medium. However, these errors are minimal and sporadic, and the DeadInterval parameter assures that minor network glitches will not contribute to greatly exaggerated rumours about a device's untimely demise.

3.3.3 CPU Utilization

The CPU of the AreaController is not placed under significant load during execution of the protocol. The highest utilization is at startup, when the VM is constructing objects and reading configuration files. During normal operation, the CPU is utilized about two percent for each request.

The CPU utilization of the VM's running the devices depends on the computation being executed. However, a device running nothing but the default PCXClient implementation contributes almost no overhead (due to no state being transferred).

3.3.4 Memory Consumption

The memory consumption of the current AreaController is about eight megabytes distributed among fifteen threads.

The memory footprint of the default PCXClient implementation is about 200kb.

3.3.5 Migration & Replacement Time

In the described testing environment, the RECOVERY phase happens quite fast. The transfer of the PacMan game to the Runner device occurs in an average of 3 seconds for ten trials. The transfer of the PacMan game, including failure detection, is dependent on the HelloInterval and DeadInterval parameters. The

transfer of the Runner application to the PacMan device happens in less than two seconds over ten trials.

3.4 Additional Requirements

Three important results are the realization of additional requirements for survivable networks of embedded systems: the development of a clear, concise, and powerful policy language, the need to recognize and prevent "failure chaining", and the need to detect Byzantine failure, perhaps through some peer-based mechanism.

4. Conclusions and Future Work

The PCXSES protocol is a general solution to the problem of process migration and fault tolerance in embedded networks. Successive transformation of the protocol for performance and security is anticipated. Most notably, a challenge-response protocol will be integrated into the GOOD MORNING phase. Alternatively, the protocol can be implemented over SSL/TLS to provide integrity and confidentiality. However, adding integrity, authentication, and confidentiality to the protocol inherits all the problems of key distribution (and refreshment) for a public key based infrastructure.

In addition, since the protocol currently assumes a common virtual machine layer on each device (to simplify the restart of processes), the protocol will be adjusted to account for different hardware targets. Perhaps standardization of lifecycle methods (in addition to saving state variables) can help achieve some level of granularity by presenting well-known hooks into object code, thus making transfer of a program counter unnecessary. The current state transfer mechanism (SUI) will be upgraded to include a translation phase between hardware targets.

5. References

[1] A. Downey and M. Harchol-Balter. "A note on 'The Limited Performance Benefits of Migrating Active Processes for Load Sharing'," University of California at Berkeley Technical Report. UCB/CSD-95-888, November 1995

[2] A. Keromytis, S. Edwards, V. Prevelakis, and M. Hicks. TC: Open and Survivable Embedded Systems (OASES). NSF Grant Proposal and Project Summary. 2002.

[3] Barak A. and La'adan O., The MOSIX Multicomputer Operating System for High Performance Cluster Computing , *Journal of Future Generation Computer Systems*, Vol. 13, No. 4-5, pp. 361-372, March 1998.

[4] D. Eager and E. Lazowska and J. Zahorjan: The Limited Performance Benefits of Migrating Active Processes for Load Sharing. In *Conf. on Measurement & Modelling of Comp. Syst., (ACM SIGMETRICS)*, May 1988, pages 63--72.

[5] J. M. Smith. A Survey of Process Migration Mechanisms. *ACM Operating Systems Review*, *SIGOPS*, 22(3): 28-40, 1988.

[6] L. Lamport, R. Shostak, and M. Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems*, Vol. 4, No. 3, July 1982, Pages 382-401.

[7] M. Hicks. Dynamic Software Updating. PhD thesis, Department of Computer and Information Science, University of Pennsylvania, August 2001.

[8] P. Smith and N. C. Hutchinson. Heterogeneous Process Migration: The Tui System. Software - Practice and Experience 28(6): 611-639, 1998.

[9] S. Edwards, L. Lavagno, E. Lee, and A. Sangiovanni-Vincentelli. Design of Embedded Systems: Formal Models, Validation, and Synthesis. *Proceedings of the IEEE* 85(3): 366-390, March 1997.

[10] R.J. Ellison, D. Fisher, R.C. Linger, H.F. Lipson, T. Longstaff, and N. R. Mead. Survivable Network Systems: An Emerging Discipline (CMU/SEI-97-TR-013) Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1997.

[11] W. Du and M. J. Atallah. Secure Multi-Party Computation Problems and Their Applications: A Review and Open Problems. In *Proceedings of the New Security Paradigms Workshop*, pages 13-22, Cloudcroft, New Mexico, 2001.

[12] Y. Artsy and R. Finkel. Designing a Process Migration Facility: The Charlotte Experience. *IEEE Computer* 22(9): 47-56, September 1989.

[13] Y. Zhang, H. Vin, L. Alvisi, W. Lee, and S. K. Dao. Heterogeneous Networking: A New Survivability Paradigm. In *Proceedings of the New Security Paradigms Workshop*, pages 33-39, Cloudcroft, New Mexico, 2001.

[14] http://www.mosix.org/ The official MOSIX website.

[15] The source and object code and associated documentation for the sample PacMan and PCXSES implementation is available at http://www.cs.columbia.edu/~locasto/projects/pcxses/

[16] http://www.ietf.org/rfc/rfc2328.txt The OSPF version 2 RFC.