M.S. Project Report

Jini on the Control Area Network (CAN): A Case Study in Portability Failure

Meredith Beveridge

Department of Electrical and Computer Engineering

Carnegie Mellon University

Phil Koopman, advisor

March 2001

Jini on the Control Area Network (CAN): A Case Study in Portability Failure

Meredith Beveridge

Carnegie Mellon University

Electrical and Computer Engineering Department

mb5@ece.cmu.edu

Abstract

The Robust Self-Configuring Embedded Systems (RoSES) project seeks to achieve graceful degradation through field reconfiguration. To accomplish this goal, systems must automatically reconfigure in the face of nodes failing, being replaced by inexact spares, or being upgraded. Thus, a "plug-and-play" run-time infrastructure is needed to allow nodes to come and go. We investigated middleware options and determined that Jini most closely matches our needs for spontaneous networking, but upon further examination we discovered that Jini makes deep assumptions about using TCP and UDP. This is appropriate for the Internet-enabled devices that the Jini designers envisioned, but distributed embedded systems such as automobiles employ the real-time, reliable data transmission supplied by the Control Area Network (CAN), rather than TCP. Jini strives for platform-independence, but it required extensive re-engineering to use Jini on CAN. Modifying Jini to use CAN provided these insights into designing truly platform-independent products: consider every piece of the system as changeable; maintain appropriate abstraction; be creative imagining application domains; and work through the details of an example for compatibility on other current platforms.

1. Introduction

This paper discusses how we chose Jini as a middleware for the RoSES project, the struggles encountered in porting Jini to the Control Area Network (CAN), the message-passing strategy to successfully get Jini functioning on CAN, and resulting heuristics for designers seeking to develop a platform-independent product.

1.1 Description of RoSES

The goal of the Robust Self-Configuring Embedded Systems (RoSES) project is to create inherently robust, flexible, maintainable, distributed embedded systems that allow for graceful degradation through field reconfiguration [6]. Rather than using a static configuration designed in the factory, such a system would automatically reconfigure to adjust for failed, upgraded, or inexact spare components (both software and hardware). We envision a system of "smart" sensors and actuators connected to an embedded real-time network, where every sensor acts as a "server" to any node desiring its functionality, as shown in Figure 1.

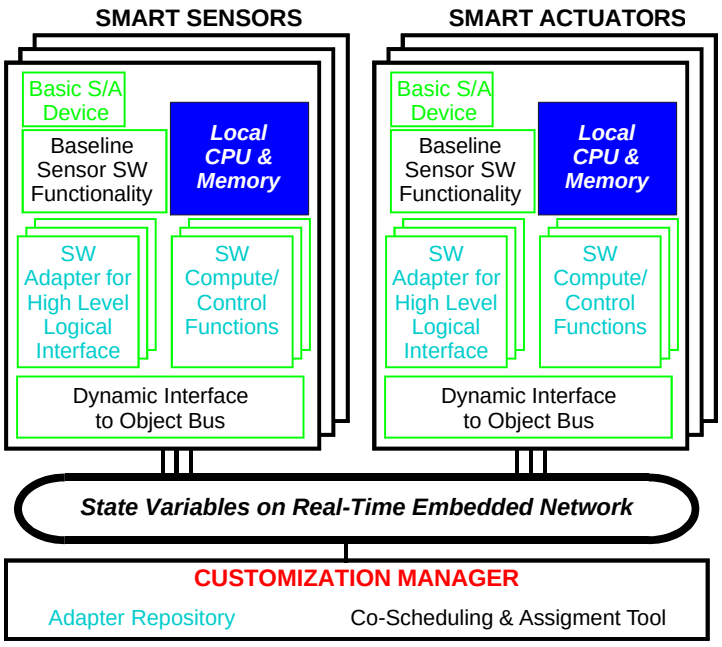

Figure 1: RoSES Architecture

1.2 The need for a RoSES infrastructure

RoSES requires some sort of “plug-and-play” infrastructure to facilitate the dynamically changing configuration caused by nodes appearing and disappearing. It is undesirable to dispatch a human administrator every time a change occurs in a system such as an automobile, so the infrastructure must allow nodes to discover the presence or absence of other nodes at run-time.

1.3 Structure of the paper

Section 2 describes the selection of Jini as the preliminary RoSES infrastructure. Section 3 discusses the special needs of distributed embedded systems that conflict with the current Jini implementation. Section 4 details the problems and solutions in applying Jini to the Control Area Network. Section 5 provides the heuristics gleaned from this portability failure. Section 6 examines related work, and Section 7 concludes the work.

2. Exploring infrastructure options: how we chose Jini

Eventually, RoSES needs middleware that meets all the needs of an embedded system, such as real-time guarantees, minimal resource usage, and no single point of failure. To determine the problems that RoSES must address, we sought an existing off-the-shelf solution with open source and proven experience in industry.

The two most popular, widely-accepted, and well-supported middleware technologies currently available are the Common Object Request Broker Architecture (CORBA) developed by the Object Management Group (OMG), and Sun Microsystems’ Jini.

2.1 CORBA

The Object Management Group (OMG) developed the Common Object Request Broker Architecture (CORBA), a middleware of well-defined, platform-independent interfaces for connecting heterogeneous

systems [8]. For large, client/server systems with large amounts of memory, CORBA is an excellent candidate. This resource-intensive paradigm does not work well with an architecture such as RoSES where every node is a "server," however. The nodes of an embedded system must be as small and cheap as possible, so resources are scarce. While OMG has developed real-time, fault-tolerant, and "minimal" (embedded) specifications for CORBA, embedded systems are not their primary target, so these efforts are slow-moving and have low levels of support. Currently, CORBA's real-time capabilities are immature at best: real-time capability is currently limited to scheduling operations on the server and adding priorities to interfaces, with no notion of system determinacy [2]. Thus, CORBA will not be well-suited for distributed, sensor-intensive embedded systems such as automobiles for quite some time.

2.2 Jini

Sun Microsystems developed Jini to provide platform-independent, spontaneous federated networking built on Java and Remote Method Invocation (RMI) [20]. The developers of both Jini and Java itself had distributed, embedded systems in mind [17], which seemed promising. In a Jini *community*, services autonomously discover other services as they become available or unavailable. Code can be downloaded dynamically to allow "clients" to use the service, eliminating the need for configuration. A centralized controlling authority is unnecessary for clients to exchange messages, eliminating a single point of failure.

The dynamically downloadable code, or *service proxies*, are stored in a *lookup service*. Nodes must first find the lookup service via the *discovery protocol*, then register their proxies with the lookup service. To find other nodes, they perform a *lookup* by sending a *template* to the lookup service. If the lookup service contains a proxy that matches the template, the proxy is returned to the requestor. The requestor can then communicate directly with the matching node via the downloaded proxy. Two additional features enhance automatic configuration: the lookup service sends out periodic *announcements* to

advertise its presence on the network, and nodes can sign up with the lookup service to be notified when other nodes appear or disappear.

The promises of spontaneous networking, platform-independence, and design for embedded systems suggested Jini would be a good fit for a RoSES infrastructure. Thus, we decided to see how much it could really provide.

3. Using Jini for distributed, embedded systems

At first glance, the limitations of Jini involved only its implementation: it was written in Java and supported only Ethernet communication.

3.1 Java on an 8-bit microprocessor??

A notable barrier for embedded systems is the size of the Java Virtual Machine (JVM). Sun's "embedded" virtual machine, KVM, has a footprint of only a few hundred kilobytes, but KVM does not currently support all of the Java capabilities required by Jini [9, 14]. Progress is also being made by groups outside of Sun, including the Jbed project [16]. We hope that the footprint of the JVM will not get any larger while chip resources continue to improve, so in the future a full JVM may fit on an embedded system.

Doubts about real-time capabilities still hinder Java's use on embedded systems, however. How can real-time scheduling be accomplished with Java's garbage collection mechanism? Can end-to-end timing be determined? What about efficiency and fairness? These issues are important, but at this point in the project we were primarily concerned with achieving a working middleware, so these issues were left for future work.

3.2 Original Jini implemented for Ethernet

Besides being written in Java, the original Jini implementation was written for devices communicating over the Internet with Ethernet, using TCP and UDP. This meant we would have to modify the Jini implementation, because real-time, safety-critical, distributed embedded systems do not use Internet protocols, primarily because Internet protocols do not provide deterministic message delivery times to satisfy real-time guarantees. The protocol of choice for embedded systems like automobiles is the Control Area Network (CAN), developed by Robert Bosch GmbH [11].

3.2.1 Description of the Control Area Network

CAN is a reliable, real-time protocol that implements a multicast, data-push, publisher/subscriber model [13]. CAN's messages are short (data payloads are a maximum of 8 bytes; headers are 11 or 29 bits); it is distributed, so there is no centralized master or hub to be a single point of failure; and it is flexible in size. Its real-time features include deterministic message delivery time and global priority through the use of prioritized message IDs.

Bus arbitration is accomplished in CAN using bit dominance, a process where nodes begin to transmit their message headers on the bus, then drop out of the "competition" when a dominant bit is detected on the bus, indicating a message ID of higher priority being transmitted elsewhere. This means bus arbitration does not add overhead because once the bus is "won," the node simply continues sending its message. Because there is no time lost to collisions on a heavily loaded network, CAN is ideal for periodic traffic. Lastly, CAN's reliability features are suited for typically harsh embedded environments.

3.2.2 CAN vs. Internet protocols

CAN was designed for a typical embedded system with periodic, bursty traffic, where every node transmits at regular intervals. Ethernet, on the other hand, was designed for aperiodic, light traffic and a low number of active transmitters. More significantly, CAN was designed to achieve tight, real-time

schedules, unlike Ethernet. Ethernet's Carrier Sense Multiple Access/Collision Detection (CSMA/CD) protocol, the "back off and retry later" approach to collision avoidance, produces an unpredictable message delivery time undesirable for tightly-scheduled real-time systems. In fact, individual message latency is completely unbounded; the minimum guaranteed latency is infinite. The usual solution for a heavily loaded Ethernet network is a switched hub, but an embedded system such as an automobile cannot afford that added weight, size, cost, cabling, and single point of failure.

TCP, the Transport Control Protocol usually employed by Ethernet and other common Internet protocols, is also a poor fit for embedded systems. TCP uses a 20-byte header, posing a strain on resource-constrained systems that send frequent, short messages. Its fragmentation and packet routing schemes produce variability that conflicts with necessary real-time message delivery guarantees, and the only notion of priority is a one-bit "Urgent" flag.

4. Porting Jini to the Control Area Network

We were not overly concerned that Jini's original implementation was intended for Internet-enabled devices, because the Jini inventors were striving for platform-independence. Jini inventor Bill Joy stressed the portability of Jini when asked, "What is the problem that Jini is trying to solve?" His answer was, "If you have two programs talking to each other, even the simplest incompatibility is really inconvenient" [4]. Likewise, Jini's lead architect Jim Waldo wrote that the problem with previous middleware attempts is that they were protocol-centric, making them inflexible to protocol changes [19]. The major goal of Jini was to "raise the level of abstraction of distributed programming from the network protocol level to the object interface level" [18]. With this encouragement from the inventors themselves, we thought it would be simple to implement Jini on top of CAN.

4.1 Portability failure

When I attempted to implement Jini atop CAN, it became painfully obvious that the Jini designers did not maintain their intended abstraction level: an examination of the Jini specification reveals features specific to TCP and UDP have crept into the "object interface level" where they do not belong, even according to Jini's inventors. While the design does not prevent the use of other protocols, it does impose unnecessary struggles in porting Jini to another network protocol. The four most glaring examples of this design error are the TCP-only identification scheme, the message size definition, the reliance upon RMI, and the unicast/multicast distinction.

4.1.1 TCP-specific identification

When Jini's protocols were defined, four distinct types of messages were created for discovering and announcing a lookup service, as shown in Figure 2.

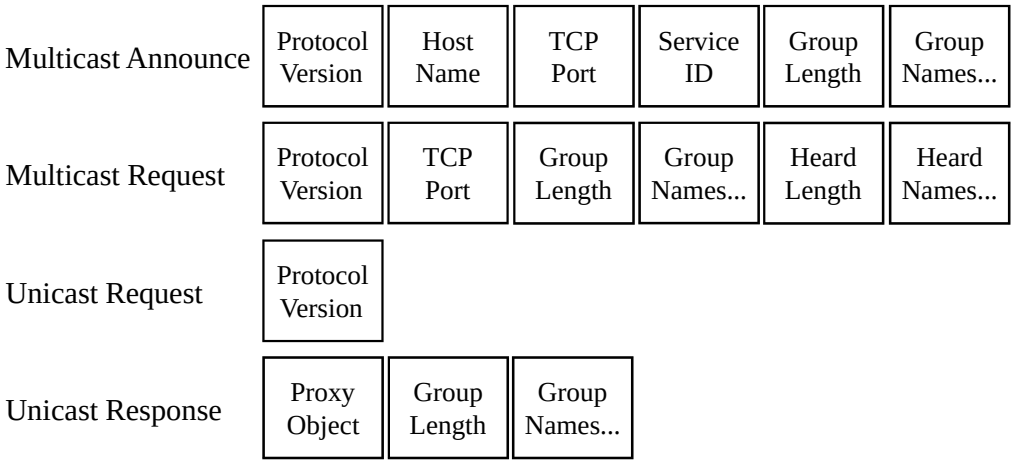

Figure 2 - Jini's four message definitions

As can be seen, two of the messages include fields for hostnames and port numbers. The designers were envisioning that these messages would be packed into datagram packets and sent over UDP, and thus would need the hostnames and port numbers for further TCP communication. The other two

messages would be sent via streams over TCP sockets, and thus no hostnames or port numbers needed to be included within the messages.

Not every wire protocol uses alphanumeric hostnames, integer port numbers, and socket-based communication. At the very least, this is inefficient use of message space, requiring that unused large message fields consume precious bandwidth. Worse, wire protocols that differ significantly in their identification schemes must devise an additional mechanism for indicating appropriate receiver and transmitter information.

A truly platform-independent design would not include this type of network protocol detail in the "object interface level." The Jini designers instead could have defined some type of standard header on every message that allowed maximum flexibility for providing whatever types of information might be desired by the specific wire protocol on which the messages are implemented. In Java, the most generic way of designing this would be to include an identification object with each message. Each implementation could then design a specific subclass of this object to include the relevant fields for that protocol, as shown in Figure 3.

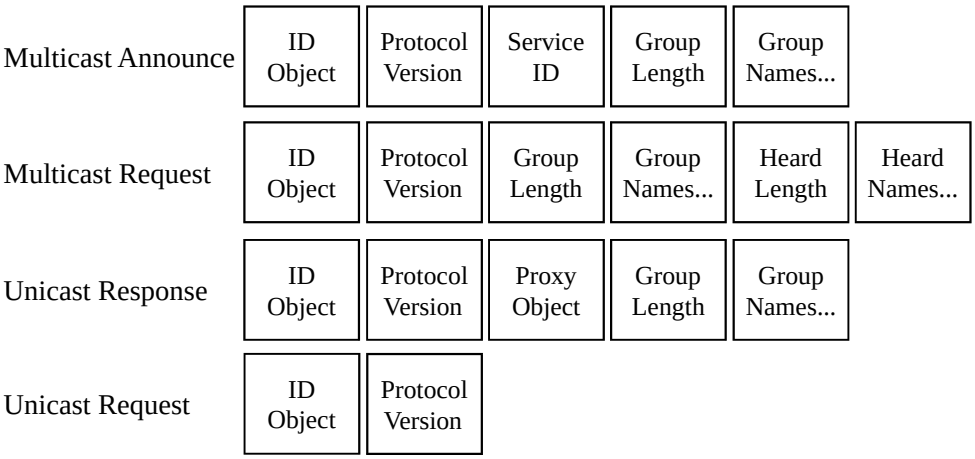

Figure 3 - Generic identification objects added to Jini message design

(To maintain a uniform design, I added the “Protocol Version” field to the Unicast Response message. This field is not included in the ID object because it is a Jini attribute; it does not influence the protocol used to send the messages). In this way, the ID object could contain the data needed by the specific protocol, such as priorities and sender/receiver information, and not pay the overhead of sending useless fields only needed by TCP-like protocols.

4.1.2 Message size definition

Including generic ID objects as headers would mean that the Jini designers would have no control over the lengths of the messages. (Jini implementors could achieve a guaranteed length by designing their ID objects appropriately, however). This should not be a concern for the Jini designers, since their goal is simply to define a platform-independent discovery protocol. However, the Jini designers made another design choice based solely on their assumption of TCP: they decreed that all messages shall fit within one UDP packet of 512 bytes. If data exceeds this size, it is fragmented and sent in multiple Jini messages. Other protocols, with different message sizes and therefore different fragmentation needs, are forced to send additional Jini messages unnecessarily. This is a great optimization for efficiently sending Jini messages via UDP, but this decision should have been made by those implementing Jini on top of UDP, not by the Jini designers themselves. This particular shortcoming is not catastrophic, but it does cause unnecessary inefficiency and software complexity for implementations on other wire protocols.

4.1.3 RMI

Once discovery has been accomplished between Jini services and the lookup services, communication transfers entirely to RMI. RMI is a powerful tool for distributed applications accessing methods on other machines, but it is also implemented solely using TCP sockets. While Jini services may choose their preferred method of communication after discovering each other, they must use RMI for all Jini communication after discovering the lookup service: registering their proxies, discovering other services, signing up for event notifications, receiving event notifications, and managing service and event leases.

Thus, some sort of platform-independent method of performing this significant portion of Jini is also necessary. Since the initial lookup service discovery was accomplished successfully with message-passing, including downloading the lookup service's proxy (a serialized Java object), it seems reasonable to continue executing Jini features with message-passing instead of switching to RMI. If services wish to continue communication with each other using RMI, they are still free to do so; this is outside of Jini's scope. Perhaps additional standard, platform-independent messages should be defined to complete the protocol, as shown in Figure 4:

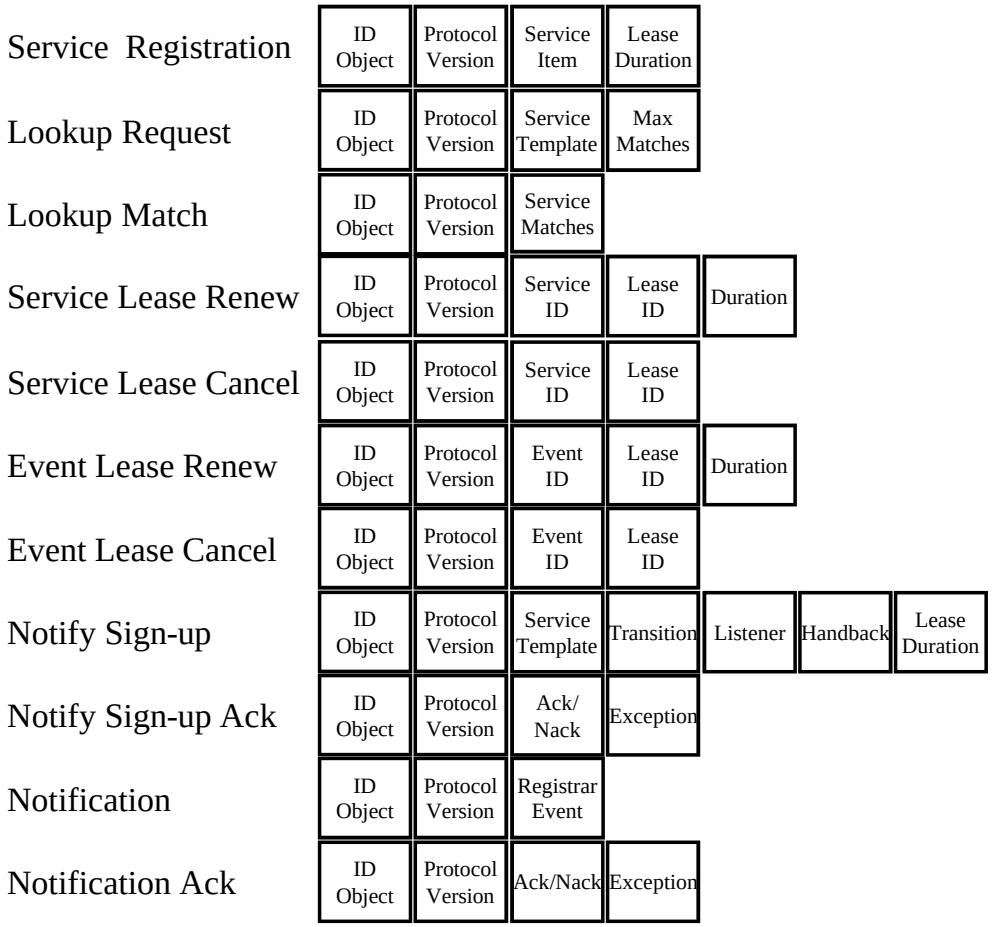

Figure 4 - Additional Jini messages

In this way, all of the Jini-provided services could be accomplished independent of the wire protocol beneath. Further communication could be continued in any manner desired by the implementors, as in the original Jini design.

4.1.4 Unicast/Multicast distinction

The last deficiency encountered in Jini's claim of platform-independence is the definition of unicast and multicast messages. In optimizing for TCP, Jini's current design sends a few multicast messages and then proceeds with unicast communication. But other wire protocols might be unicast only, multicast only, broadcast only, or any combination. For instance, CAN is a broadcast bus. Unicast can be emulated on CAN and may still be useful in some cases, but it is wasteful. In implementing Jini on CAN, it became apparent that unicast communication was not needed at all, since the same functionality could be accomplished with the equivalent multicast request and response.

This overlap of “unicast request” and “multicast request” demonstrates the lack of an abstraction level distinction in the Jini design. An “announcement,” “request,” and “response” do not change when implemented on unicast, multicast, or broadcast networks; their function is the same regardless of the underlying protocol. The lookup service is looking for discovery requests and does not care if they are sent via unicast or multicast request; similarly, the discoverer does not care if the lookup service’s response is sent via unicast or multicast, but only that it receives the response. At the “object interface level,” a unicast/multicast distinction is irrelevant and therefore should not be specified in the Jini message protocols. This information could be included in the above-proposed ID object if the implementor so desired, or the implementor could simply define it to be one way or another, but the messages themselves should only specify details pertinent to their functionality, and not the specifics of layers underneath. This would further change the Jini message definition as shown in Figure 5.

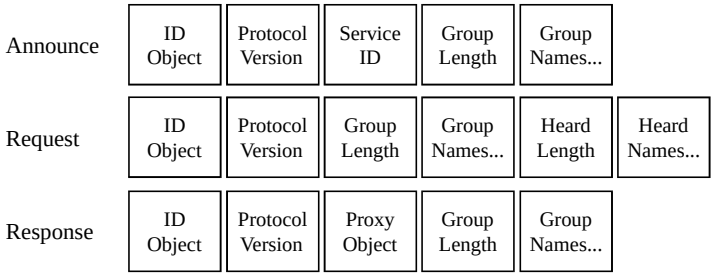

Figure 5 - Generic Jini messages without unicast/multicast specified

4.1.5 Possible platform-independent design

The combination of the above changes would result in a platform-independent Jini message-passing scheme similar to Figure 6:

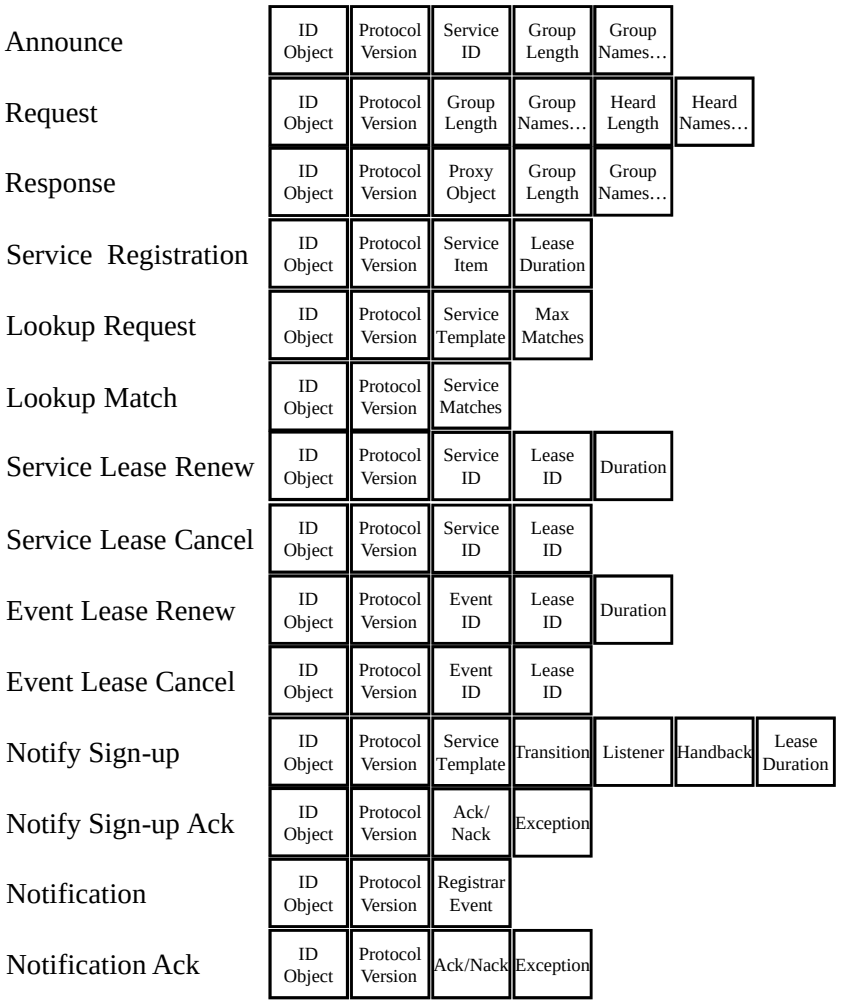

Figure 6: Resulting platform-independent Jini messages

4.2 Workarounds

With such a truly platform-independent design, we would have been able to directly port Jini to CAN and easily achieve a plug-and-play infrastructure for RoSES. Unfortunately, Jini was not designed this way. Our goal was not to redesign Jini, however: we just needed a testbed to use for understanding graceful degradation. This section describes the solutions created to get the original Jini working on top of CAN.

6.2.1 TCP-specific identification

To address the problem of insufficient ID information for non-TCP protocols I designed an ID generator that combined a unique node ID and constants defined for each type of Jini message into a unique 29-bit (extended) ID. Further ID information was written in the payload data. The scheme is more clearly explained in Figure 7.

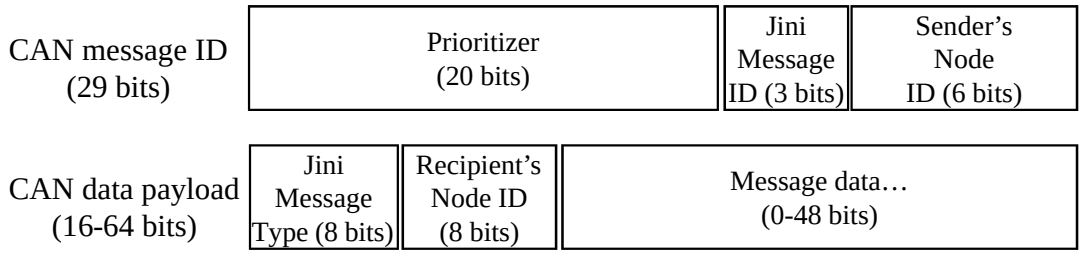

Figure 7 - Basic ID scheme

The "prioritizer" is used to determine the priority of Jini messages over other messages on the bus (recall that CAN arbitration is accomplished using the message ID itself). In this way, normal CAN bus traffic can continue in the presence of Jini message traffic. This was designed to be flexible for various situations of the system; for instance, at start-up you may want Jini messages to have high priority so the nodes can discover and register quickly and get on with doing work, and after a sufficient time you may want to make the subsequent Jini communication less important than the other bus traffic.

The Jini message ID is simply a unique ID for each type of Jini message so that receivers of multicast messages have a way of identifying the messages they're looking for. All of the messages shown above in Figure 6 could be given separate CAN IDs, but I combined some of them to conserve CAN IDs. (This does not prevent each message from being defined separately as shown in Figure 6; their individual definitions are simply implemented in the following manner to conserve CAN IDs). Most messages use either a "server request" ID or a "server response" ID, where "server request" refers to a service's request to the lookup service (including registration, lookup, lease management, etc.), and "server response" refers to the lookup service's response to a service's request. (A separate set of IDs is assigned for multicast requests, "unicast" responses, and event notifications. These messages could all be classified as "server request" or "server response," but they are separated for optimization purposes). These message IDs are shown in Table 1:

Message Type	ID
"Unicast" response	0
Multicast request	3
Server request	4
Server response	5
Event notification	6
Event notification response (ack/nack)	7

Table 1: Jini message IDs

The sender's node ID is included in the CAN header to ensure unique CAN IDs. In the case of messages designed to be received by everyone, this field is simply ignored, and listeners listen to the entire range for that type of message (i.e., [prio][msg type][0] - [prio][msg type][63]). The node IDs could be any 6-bit numbers as long as each node had a unique number; for convenience, I used the last byte of statically assigned, sequentially numbered IP addresses.

An additional byte is written in the data payload to distinguish between the many different types of messages grouped as "server request" or "server response." At the expense of payload data, this avoids consuming a larger set of CAN message IDs, and allows recipients to listen for just one or two message types, rather than large ranges of messages.

Since IDs are labeled only with the sender's node ID and not the recipient's, the sender of the message puts the intended recipient's node ID as the second byte of the data payload. The recipient can then check to make sure the response was intended for it. This data could have been included in the message ID instead, but I chose to conserve CAN message IDs.

6.2.2 Message size definition

Since I replaced the UDP communication with a stream that sent all data directly to the CAN message fragmentation algorithm, I did not use the Jini-defined message fragmentation at all. Clearly, the message size was something that Jini did not even need to define.

6.2.3 RMI

I originally attempted to implement Java "socket factories" to perform RMI communication over CAN. The factory approach was attractive because it would define a default factory for all sockets created, so that all code that invokes a socket would create the specialized socket defined by the factory. (In my case, the specialized socket would convert TCP input to CAN messages, and CAN input to TCP streams). However, these socket factories were not intended to be used to swap wire protocols; the designers only envisioned such uses as firewalls and cryptography. I believe it may have been possible to eventually achieve RMI over CAN using socket factories, but I abandoned this effort after weeks of re-implementing handshaking and other TCP-socket-specific communication unnecessary for CAN.

I thus decided to attempt the message-passing approach described in Section 4.1.3, employing Java's MarshalledObject and serialization features extensively. Several different objects implementing proxies

and leases used RMI calls, so I had to find them all and convert the calls to serialize data (even whole objects) into CAN messages and then reconstitute the objects on the other end. The Jini concept of downloading proxies for performing implementation-specific communication proved invaluable in this solution, since the proxies encapsulated all the details of serializing/deserializing data, constructing CAN messages, and so on. The application is oblivious to the change, still calling standard methods defined in proxies' interfaces, as cleanly as if they were communicating via RMI behind the scenes.

This approach was very successful. Implementation took only a week.

6.2.4 Unicast/Multicast distinction

Since CAN is broadcast, the concept of "unicast discovery" is irrelevant. It could be emulated, but it would be difficult to ensure unique CAN message IDs at all times. Instead, multicast discovery can be used, and unicast discovery abandoned entirely. Since multicast announcements are used solely to invoke unicast discovery from other nodes, the multicast announcement is therefore also unnecessary. This resulted in using only two of the original four Jini messages, and multicasting the “Unicast Response” message.

6.2.5 Result of workarounds

The result of the four solutions described above looked like Figure 8. Implementing Jini on CAN in this way allowed us to approximate the RoSES architecture, as shown in Figure 9.

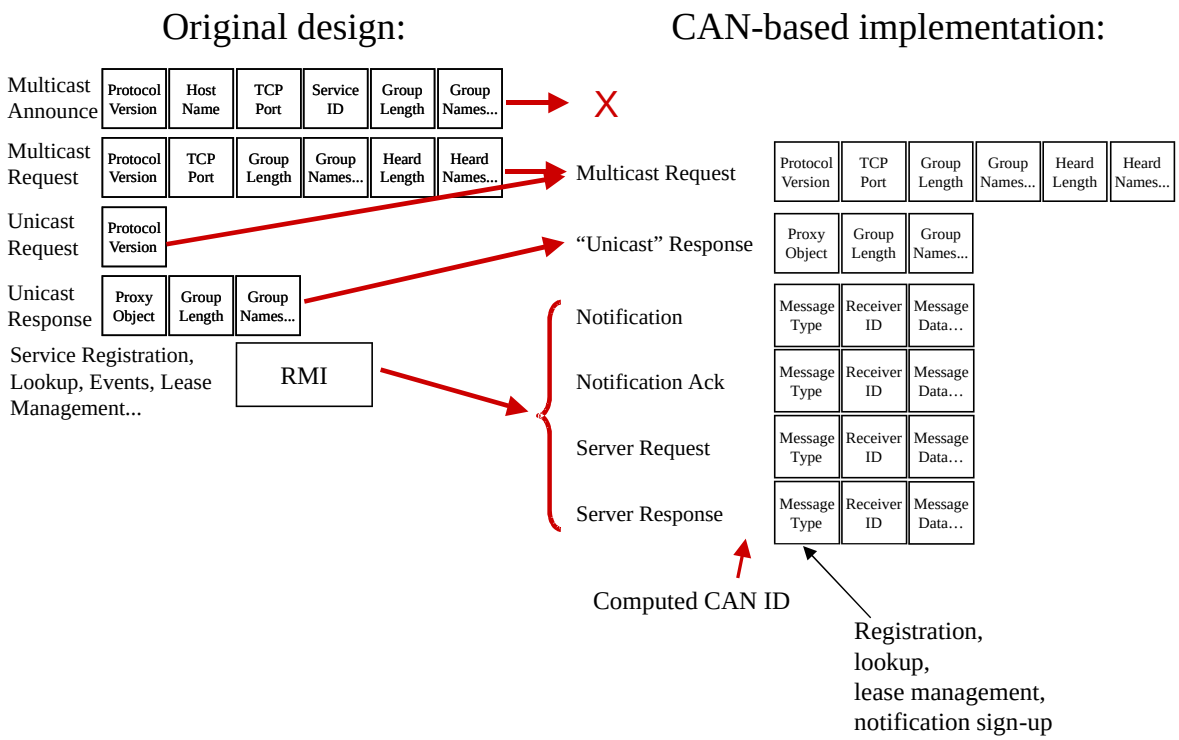

Figure 8: Original Jini modified to work on CAN

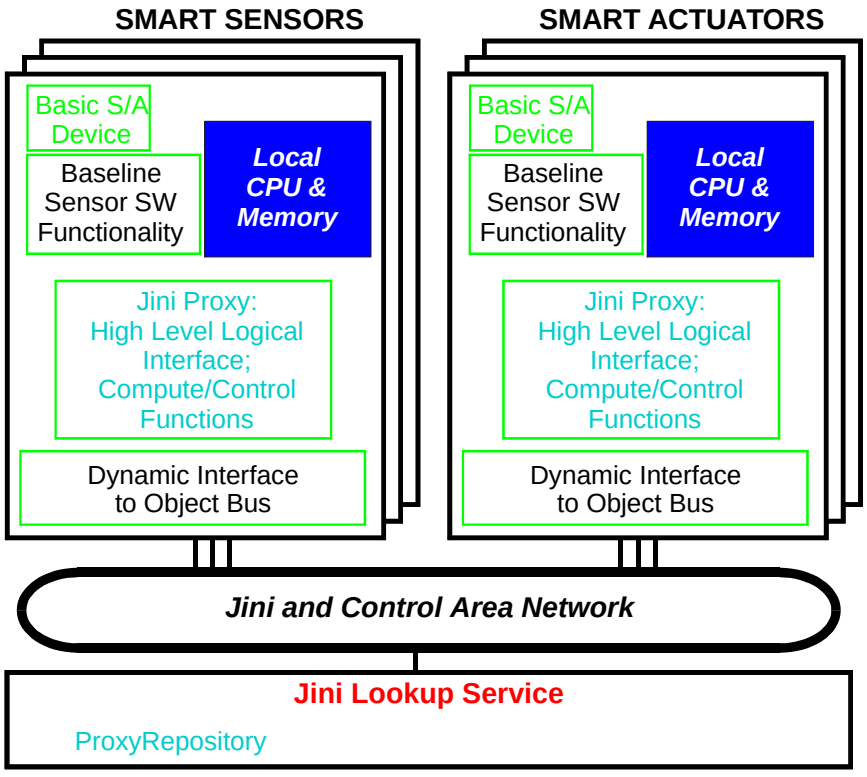

Figure 9: Approximated RoSES architecture

5. Heuristics for platform-independent design

From this experience we were able to demonstrate how one would implement graceful degradation with current technology (by using the discovery mechanism of Jini running on CAN), and determine what problems our future work must address. In the process, I also gleaned insight into how *not* to design for platform-independence. Based on my experience I propose the following list of heuristics for any designer who seeks to achieve platform-independence.

1. Think very broadly: do not assume that *any* part of your imagined scenario will remain the same for all users.

Jini lead architect Jim Waldo asserts that the most important computing component is now the network [17]. Yet with all their focus on the importance of the network, the Jini designers forgot that the hardware is not the only variable: TCP is not universal either!

2. Design to what *your* product does, independent of underlying hardware or software layers. I.e., stay within your targeted abstraction level.

The intrusion of unicast/multicast, port numbers, hostnames, and message sizes into the "object interface level" does not maintain a clean abstraction necessary for platform-independence. The Jini specification should stick to *what* the Jini messages are accomplishing, not *how* they are accomplishing it.

3. Never assume your product will only be used the way you imagined it.

Jini can be used for more than connecting consumer gadgets. Apparently we are not the only ones to think of this; a Sun employee himself suggested Jini as a solution for control systems [10]. You cannot imagine every possible use of your system, but adhering to a good abstraction principle should help.

4. You cannot imagine everything that will be invented in the future. But if your product is to be platform-independent, at least consider how your product would actually work on other current systems.

Sun employees were surprised by my struggles. Apparently, while touting platform-independence, they had never thought through how one would swap network protocols. Not only that, but they were completely unaware of how immensely different various wire protocols can be. CAN was invented over a decade before they designed Jini; they just did not think to look at other protocols.

6. Related Work

6.1 Middleware research

Many middleware solutions exist, such as Salutation, which performs Jini- like discovery but supports non-Java devices [12], and the Distributed Embedded Object Model (DEOM), designed to support distributed embedded systems [1]. Another interesting project involving remote method invocation in real-time on CAN is not addressing automatic reconfiguration issues, but it may be possible to extend it for this purpose [3]. Our goal, however, was to acquire a working middleware, port it to CAN, and identify our project's further needs; we therefore needed a middleware with well-supported open source.

6.2 Middleware on CAN

A previous attempt to put CORBA on CAN required so many changes that the result looked very little like CORBA [5]. Current work applying Jini to embedded systems has been focused on using the "surrogate architecture," which uses a JVM-capable device as a gateway between the CAN devices and the rest of the Jini community [7]. Surrogates are useful for remotely diagnosing the CAN system over the Internet, for example, but we want our CAN nodes to form a Jini community themselves, so that each CAN node can exploit Jini's self-configurability. The developers on the JINI-USERS mailing list, which include some of the original Jini developers, knew of no attempts to implement Jini on any protocol besides TCP/IP [15].

7. Conclusions

For the Robust Self-Configuring Embedded Systems (RoSES) project to succeed in its goal of graceful degradation through field self-reconfiguration, an appropriate run-time infrastructure must be in place to facilitate the "plug-and-play" functionality required for nodes to easily come and go from an ad hoc, distributed network. The embedded, real-time network of choice for automobiles and other distributed, safety-critical systems is the Control Area Network (CAN). CAN provides reliable, real-time, multicast, decentralized communication, but it in itself does not have the capability of accomplishing the RoSES "plug-n-play" goal. We investigated various middleware technologies to supply a run-time infrastructure on top of CAN, ultimately choosing Jini.

In the process of porting Jini to CAN, we discovered that the very design of Jini itself made assumptions about the use of TCP and UDP, including choices of packet sizes and message fields. This did not prevent the porting of Jini to CAN, but imposed a significant source of inefficiency and messiness in substituting wire protocols. Designers who wish to produce a platform-independent product should maintain an appropriate abstraction level to avoid entangling details of the underlying protocols into the product. One possible “sanity check” would be to imagine how one would use the new protocol on diverse, currently existing systems. Designers must consider every part of the system that could be implemented in a different way from that originally supposed, and avoid the handicap of thinking only of a single target scenario.

8. Acknowledgments

This work was supported by the General Motors Satellite Research Laboratory at Carnegie Mellon University, Robert Bosch GmbH, the NSF fellowship program, and the Intel IMAP fellowship program. I would also like to acknowledge Bill Nace, Keith Thompson, Greg Frazier, Geoffrey Clements, and Mike Bigrigg for their invaluable assistance in this work.

References

[1] Bacellar, L.F., and Upender, B.P. "A Dependable Distribution-Transparent Remote Method Invocation Model for Object-Oriented Distributed Embedded Computer Systems," in *Proceedings of the First International Symposium on Object-Oriented Real-Time Distributed Computing,* Kyoto, Apr. 1998, p. 467-76.

[2] Bigrigg, M., OMG member, interview Aug. 2000.

[3] Kaiser, J., and Livani, A. "Invocation of Real-Time Objects in a CAN Bus-System," in *Proceedings of the First International Symposium on Object-Oriented Real-Time Distributed Computing,* Kyoto, April 1998, p. 298-307.

[4] Kelly, K., and Spencer, R. "Creating One Huge Computer," in *Wired* magazine, Aug. 1998.

[5] Kim, K., et al. "Integrating subscription-based and connection-oriented communications into the embedded CORBA for the CAN bus," in *Proceedings of the Sixth IEEE Real-Time Technology and Applications Symposium,* Washington, D.C., June 2000, p. 178-187.

[6] Nace, W., and Koopman, P., "A Product Family Architecture Approach to Graceful Degradation," in *Proceedings of the International IFIP WG 10.3/10.4/10.5 Workshop on Distributed & Parallel Embedded Systems,* Paderborn, Germany, Oct 2000.

[7] Nusser, G. and Gruhler, G. "Dynamic Device Management and Access Based on Jini and CAN," in *Proceedings of the Seventh International CAN Conference,* Amsterdam, Oct. 2000.

[8] The Object Management Group (OMG). *CORBA specification version 2.4,* Oct. 2000.

[9] Pawlan, M., "Introduction to Consumer and Embedded Technologies," Sun Microsystems *Java Developer Connection,* Aug. 2000.

[10] Renner, K. “Ending the Bus Wars,” in *Sensors,* vol.16, no.5 p. 42-50, May 1999.

[11] Robert Bosch GmbH. *Control Area Network specification version 2,* Sept. 1991.

[12] The Salutation Consortium. *Salutation specification version 2.0c*, June 1999.

[13] Schill, J, "An Overview of the CAN Protocol," in *Embedded Systems Programming*, Sept. 1997, p. 46-62.

[14] Sun Microsystems. White paper: *Java 2 Platform Micro Edition (J2ME) Technology for Creating Mobile Devices*, May 2000.

[15] Thompson, K., Sun employee; Frazier, G.; and Clements, G. Online correspondence with Jini developers, July 2000 - Feb. 2001.

[16] Tryggvesson, J., et al. "JBED: Java for Real-Time Systems," in *Dr. Dobb's Journal,* Nov. 1999, p. 78-86.

[17] Veneers, B., "The Jini Vision: A glimpse into the vision behind Jini technology," in *JavaWorld*, Aug. 1999.

[18] Veneers, B., “Objects, the Network, and Jini: How Jini raises the level of abstraction for distributed systems programming,” in *JavaWorld*, June 1999.

[19] Waldo, J. “The End of Protocols”, Sun Microsystems *Java Developer Connection*, June 2000.

[20] Waldo, J. "The Jini Architecture for Network-Centric Computing," in *Communications of the ACM*, vol. 42, no. 7, July 1999, p. 76-82.