Jini Meets Embedded Control Networking: a case study in portability failure

Meredith Beveridge
Schlumberger
Sugar Land, TX, USA
meredith.beveridge@ieee.org

Philip Koopman
ECE Department
Carnegie Mellon University
Pittsburgh, PA, USA
koopman@cmu.edu

Abstract

The Robust Self-Configuring Embedded Systems (RoSES) project seeks to achieve graceful degradation through software reconfiguration. To accomplish this goal, systems must automatically reconfigure despite nodes failing, being replaced by inexact spares, or being upgraded. Jini seemed to provide the required spontaneous networking infrastructure, but turned out to make deep assumptions about using TCP and UDP. This is appropriate for the Internet-enabled devices that the Jini designers envisioned, but typical distributed embedded systems employ real-time, reliable data transmission such as the Control Area Network (CAN), rather than TCP. Object-oriented technology such as Jini is often represented as being suitable for use in real-time embedded systems. But despite Jini's goal of platform-independence, it required extensive re-engineering to function on CAN. This case study of an actual implementation of Jini on a CAN network demonstrates that the differences between general purpose and embedded systems can be more fundamental than is generally appreciated.

1. Introduction

The goal of the Robust Self-Configuring Embedded Systems (RoSES) project is to create inherently robust, flexible, maintainable, distributed embedded control systems that support graceful degradation via in-service software reconfiguration [8]. Rather than using just a static configuration established at the factory, such a system would automatically reconfigure to accommodate 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. Software adapters are loaded into each node on the embedded network to translate between architected state variables on the system network and the control and sensing capabilities of various sensors and actuators. A customization manager will be used to maximize system-level functionality by solving an optimization problem of allocating a subset of possible functionality to maximize overall utility [9]. Obviously a centralized resource manager would make the system vulnerable to a single point of failure, so a distributed implementation of that capability is highly desirable.

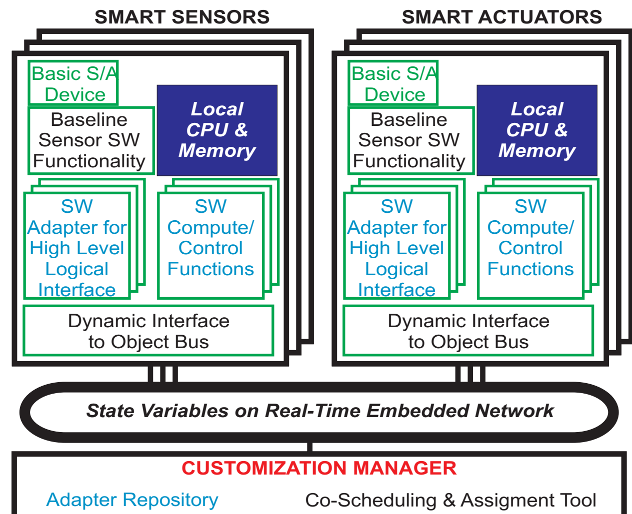

Figure 1: The generic RoSES architecture.

Creating a malleable system along these lines requires some sort of "plug-and-play" infrastructure to allow nodes to discover the presence or absence of other nodes at run-time. Dynamic reconfiguration has been accomplished on the desktop using middleware for some time. An embedded system differs from the desktop environment, however, in its need for real-time guarantees, minimal resource usage, and no single point of failure. Eventually, RoSES must have middleware that meets all these needs.

To keep our research focused on policy and architecture issues, we wished to use existing middleware technology. We therefore sought a well-tested, off-the-shelf solution with open source and proven capability. But it had to be made to work on typical embedded hardware platforms to be useful to us. Section 2 of this paper discusses how we initially chose Jini as a middleware for the RoSES project.

Section 3 describes why we chose the Control Area Network (CAN) for our testbed and some ways in which it differs from typical desktop networks. Section 4 discusses the struggles encountered in porting Jini to CAN and the message-passing strategy that was finally successful in getting Jini functioning on CAN. Section 5 presents experimental results, and Section 6 summarizes lessons learned.

2. Selecting Jini

The first step in attempting to adopt an off-the-shelf middleware product for RoSES was identifying essential embedded system constraints. The major concerns for selecting middleware are the need for a real-time networking protocol, and a general scarcity of processing power and memory in inexpensive smart sensor/actuator nodes.

2.1. Resource constraints

Unfortunately, typical middleware assumes the availability of abundant computing resources and a desktop network such as Ethernet. While these might be realistic for high-end "embedded" systems such as set-top boxes, neither assumption is valid for more prevalent embedded applications such as cars, elevators, jet engines, building environmental controls, highway systems, and railways.

Because most embedded systems have severe resource constraints, we looked for middleware that was "lean" and could plausibly fit in a $1 to $10 computer system in the reasonable future. Because embedded networks tend to have different tradeoffs than desktop networks, we also looked for middleware that would be portable beyond the network protocols found in the desktop computing world.

Java is known for its portability, so Sun Microsystems' Jini seemed most promising because it runs on Java. In fact, the developers of both Jini and Java had distributed, embedded systems in mind [16]. Although Java is still too large for many embedded systems and poses real-time challenges, these problems are being addressed by others [11,15]. Furthermore, our primary objective was to find a middleware concept that worked for RoSES prototype systems, which have loosened size and cost constraints compared to real embedded systems.

The major goal of Jini is to "raise the level of abstraction of distributed programming from the network protocol level to the object interface level" [17]. Thus it seemed an ideal way to attain platform independence in the extremely varied world of embedded networking.

Although original implementation was intended for Internet-enabled devices, the Jini inventors were specifically striving for platform-independence. Jini inventor Bill Joy stressed the portability of Jini as a central problem that he was trying to solve when he said "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 [18].

2.2. Description of Jini

Jini was developed to provide platform-independent, spontaneous federated networking built on Java and Remote Method Invocation (RMI) [19]. 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. A centralized controlling authority is unnecessary once the system is running, reducing single point failure vulnerability.

Dynamically downloadable code, or *service proxies*, is 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.

2.3. Other relevant middleware

Many middleware solutions exist, such as Salutation, which performs Jini- like discovery but supports non-Java devices [13], and the Distributed Embedded Object Model (DEOM), designed to support distributed embedded systems [1]. Another interesting project involving realtime remote method invocation on CAN is not addressing automatic reconfiguration issues [3], but it may be possible to extend it for this purpose. Our goal, however, was to acquire working middleware on CAN to create a testbed to explore our areas of interest, so we needed middleware with well-supported open source and proven capability.

3. Control Area Network (CAN)

In order to be relevant for the majority of embedded systems, RoSES must be able to operate on established embedded real-time control network protocols. CAN is the current de facto standard protocol for many embedded applications, and has stood the test of time. Furthermore, since it was specifically designed for the automotive arena, it was attractive for this testbed since the first RoSES example systems will be automotive applications.

Internet protocols are designed to provide worldwide connectivity, but CAN is optimized for operation on a closed-end control system, typically with 32 or fewer nodes. So, while Internet protocols identify nodes with a

hostname and IP address and use lengthy packet headers, CAN messages must fit most relevant header information in a standard (11-bit) or extended (29-bit) CAN header field. Additionally, bandwidth is constrained by economic and physical limitations to 1 Mbit/sec or slower.

CAN is optimized for the short, periodic messages of a typical control system, so data payloads are limited to 8 bytes. Every message is broadcast on the bus in such a way that bounded and predictable message delivery time as well as global message prioritization are guaranteed. Lastly, CAN's reliability features are suited for typically harsh embedded environments. Thus, CAN is an attractive choice for RoSES (and generally representative of embedded networks in general), but has dramatically different goals and design tradeoffs compared to desktop computers.

3.1. Previous middleware on CAN efforts

A previous attempt to put CORBA on CAN [5] required so many changes that it is unclear if the result was still "CORBA" in a practical sense. In general CORBA does not seem lean enough to be viable for RoSES in the near future. (Perhaps that will change when an embedded CORBA standard emerges, but such systems are not yet available.)

Previous work in 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 [10]. Surrogates are useful for remotely diagnosing the CAN system over the Internet, and other similar applications. However, the success of RoSES hinges on making each smart sensor/actuator a first-class citizen in the network, meaning that CAN nodes must be able 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 [14]. Similarly, attempts to find a TCP/IP implementation that runs on CAN were fruitless.

4. Fundamental portability issues

Jini's promises of spontaneous networking, platform-independence, and design for embedded systems seemed to bode well for use as RoSES infrastructure. At first glance, the limitations of Jini involved only its implementation: it was written in Java and supported only Ethernet communication. The use of Java was readily handled by buying small desktop computers as the first nodes for the testbed and arguing to Moore's Law to demonstrate long-term feasibility for embedded systems. Ethernet, however, had to be replaced with CAN to provide a credible real-time networking capability for the testbed.

However, it soon became apparent that Jini did not achieve its 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 arguably do not belong. While the design does not strictly prevent the use of other protocols, it does impose unnecessary struggles in porting Jini to another network protocol. The four most serious issues are: a TCP-centric message identification scheme, an overly restrictive message size definition, reliance upon RMI, and a unicast/multicast distinction that could have been avoided.

4.1. 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.

Multicast Announce	Protocol Version	Host Name	TCP Port	Service ID	Group Length	Group Names...
Multicast Request	Protocol Version	TCP Port	Group Length	Group Names...	Heard Length	Heard Names...
Unicast Request	Protocol Version					
Unicast Response	Proxy Object	Group Length	Group Names...			

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.

However, there are a great many protocols in use in embedded systems, and most such wire protocols use neither alphanumeric hostnames, nor integer port numbers, nor socket-based communication. At the very least, requiring large message fields that only apply to some protocol stacks is inefficient, which is a severe issue in bandwidth-constrained embedded systems. Worse, wire protocols that differ significantly in their identification schemes must devise an additional mechanism for indicating appropriate receiver and transmitter information.

To address the problem of insufficient ID information for non-TCP protocols, an ID generator was designed that combined a unique node ID and constants defined for each type of Jini message into a unique 29-bit (extended) CAN ID. Further ID information was written in the payload data as shown in Figure 3.

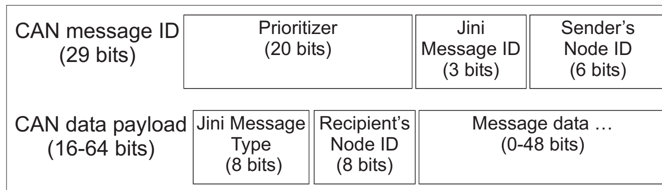

Figure 3. Jini message ID scheme

The sender's node ID is included in the CAN header to ensure a unique CAN header field (this is required by the CAN prioritized message transmission mechanism). In the case of messages designed to be received by all nodes, this field is simply ignored, and listeners listen to the entire range for that type of message (i.e., [prio][msgID][0] - [prio][msgID][63]). The node IDs can be any 6-bit numbers as long as each node has a unique number, which encompasses the needs of most embedded networking applications. A 20-bit prioritizer is maintained to support global prioritization and permit compatibility with concurrent non-Jini network traffic.

An additional byte is written in the data payload to further identify the Jini message type beyond the 3-bit identifier included in the header. 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 this design benefits from embedded system design experience that teaches conserving CAN message header bits is highly desirable.

4.2. Message size definition

An additional assumption of TCP/UDP that has crept into Jini is the requirement that all messages fit within one UDP packet of 512 bytes. If data exceeds this size, it is fragmented and sent in multiple Jini messages. While the notion of fragmentation is important, most older embedded protocols require additional fragmentation to shorter packet lengths, thus requiring duplication of fragmentation services in both the protocol interface (to get to 512-byte logical packets) and in Jini (to get to larger application packets than 512 bytes). This could cause inefficiencies even in desktop systems if native packet sizes are increased at a later date.

The message size constraint does not cause a fundamental problem for CAN, but does result in unnecessary software complexity and inefficiency, which is always an issue for embedded systems.

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

4.3. RMI

Once discovery has been accomplished between Jini services and the lookup services, further communication transfers entirely to Remote Method Invocation (RMI). RMI is a powerful tool for distributed applications accessing methods on other machines, but it is unfortunately implemented solely using TCP sockets, which are unavailable on many embedded systems. 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.

This problem was solved by implementing a message-passing approach that employs Java's MarshalledObject and serialization features extensively. The existing 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 result is that Jini applications do not see the changes needed to remove dependence upon RMI. They still call standard methods defined in proxies' interfaces, and things work just as cleanly as if RMI were in operation behind the scenes. Thus, while RMI might well have been convenient scaffolding for creating Jini, it is not essential to Jini's operation.

4.4. Unicast/Multicast distinction

The last impediment to platform-independent operation encountered in Jini was the definition of unicast and multicast messages. In optimizing for TCP, Jini's current design sends a few multicast messages and then switches to unicast communication. However, other wire protocols are unicast only, multicast only, broadcast only, or various combinations. Thus, Jini performs operations that are optimized for one platform that might well cause significant network performance problems in another system.

For instance, CAN is a broadcast bus that can use optional receiver filters to achieve multicast transmission. Unicast can be emulated on CAN and may still be useful in some cases, but is generally undesirable because it either consumes precious header bits or requires application-level coordination between the sending and receiving nodes.

But 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. Since CAN is broadcast, the concept of "unicast discovery" is not very useful in that context. It could be emulated, but it would be difficult to ensure unique CAN message IDs at all times. Instead, multicast discovery can be used in all cases, 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.

This overlap of "unicast request" and "multicast request" leads to the possibility of switching to a generic request format that would give implementations more flexibility in achieving efficient operation. An "announcement," a "request," and a "response" do not change when implemented on unicast, multicast, or broadcast networks; their functions are 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 does not seem to really belong in Jini message protocols.

4.5. End result: Jini ported to CAN

While far more painful than originally envisioned, portability problems with Jini were overcome and it was ported to a CAN hardware testbed. The significant issues that had to be resolved were:

1. CAN does not have an IP field, but that can be approximated with clever header exploitation;
2. CAN message sizes are significantly smaller than Jini message sizes, but Jini messages can be fragmented;
3. CAN does not support TCP, which is required for RMI communication, but RMI can be replaced with a message passing scheme; and
4. CAN does not have the facility for unicast messages, but messages can be broadcast and identified for single recipients via the header.

Figure 4 shows the message formats of mapping Jini onto CAN with this design approach. Figure 5 shows the resultant implementation of the RoSES architecture on the Jini+CAN testbed. The real-time embedded network was implemented with CAN, and communication of state variables and other information was implemented using Jini. A simple customization manager was implemented via a Jini Lookup Service, and the Jini ProxyRepository served as an adapter repository.

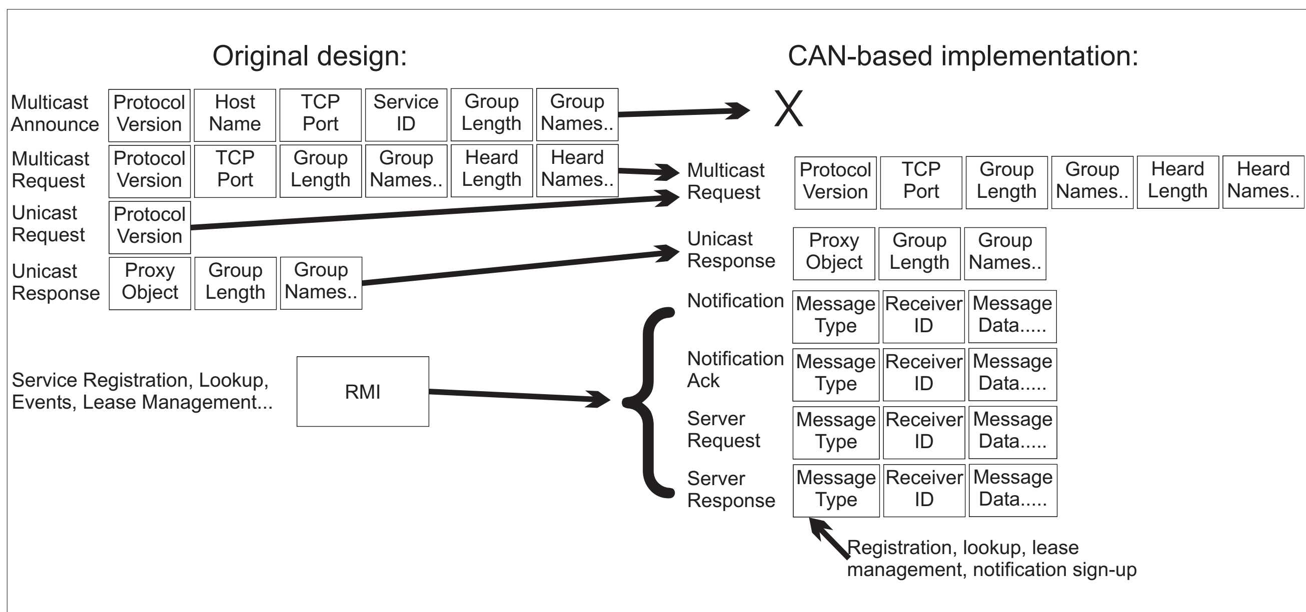

Figure 4. Mapping of original Jini messages into a CAN-based implementation

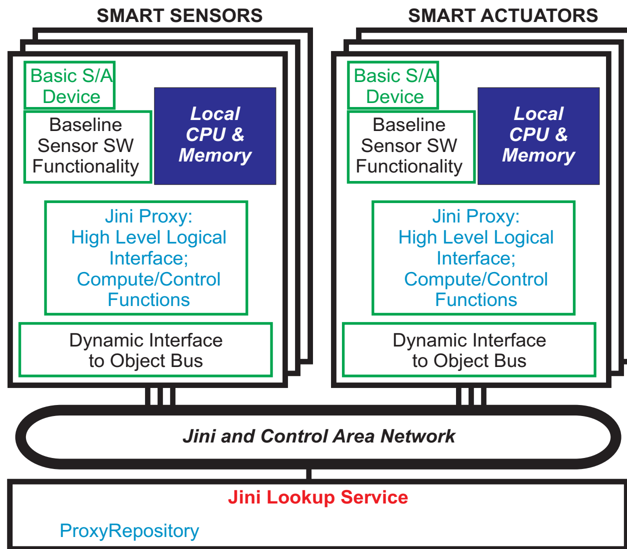

Figure 5. RoSES implemented with Jini on CAN

5. Experiences from operating a testbed

After several months of struggle (and traveling down many blind alleys in pursuit of the eventual solution discussed above), experiments were run on a RoSES hardware testbed using a simple automotive example application. The results of this experience revealed still further issues in attempting to use Jini for embedded control applications.

5.1. Slow reconfiguration

The most debilitating result of trying to make Jini work on CAN was the lack of determinacy. Even though CAN is a real-time network, that does not force software applications to behave in a real-time fashion. Jini follows a fairly relaxed approach to managing busy resources, resulting in nodes taking as long as thirty minutes just to register with the lookup service, even with no other network traffic. While this might be appropriate for metropolitan area networks and durable computing nodes, it is unsuitable for many embedded applications and for RoSES in particular. For example, waiting a half-hour between turning on the ignition and being able to drive a car is unacceptable.

An additional problem is that the amount of network traffic caused by the large message sizes is not just inefficient, but unrealistic for a network full of safety-critical messages: the large amount of data transmitted frequently results in over 255 CAN messages per Jini communication "message". The inefficiency of the entire design is restrictive for resource-scarce embedded systems: overhead of the enforced message sizes is as much as 14% of the entire message length. The bandwidth spent on IP routing information comprises as much as 33% of a Jini message. This bandwidth was wasted when we implemented Jini on a different network protocol, because the IP information is irrelevant to CAN. Further, additional bandwidth must be consumed to transmit the relevant CAN ID information.

Lease management is another issue which must be addressed to efficiently operate Jini on a real-time system. Currently, services register their proxies with the lookup service and receive a lease for a certain amount of time. If the lease is not renewed before it expires, the lookup service discards its reference to that service's proxy. For real-time systems, some nodes need to know immediately when other nodes fail, but the lookup service cannot send event notifications until the failed node's lease has expired. The leases could be set for the shortest time possible, but that would significantly increase network traffic from nodes trying to renew their leases frequently. The large number of messages would likely result in many nodes having to re-register frequently with the lookup service because of timed out lease renewal attempts, further increasing network traffic.

To get the testbed operational, an extra background task was added to query nodes at periods of several seconds and kill off leases for any that didn't respond. This method is undesirable, since it subverts the Jini system and requires knowledge of every node's processes and node IDs. But, it was expedient for getting the testbed operational and verifies the nature of problems observed with Jini. One way Jini could solve this problem is by adding a separate timer for fault detection that is independent of the lease time.

5.2. Issues with other embedded protocols

The issues we discovered in Jini are not restricted to CAN, and will probably pose similar problems on most real-time embedded systems. Jini reflects its heritage of the desktop computing world and an implicit definition of "embedded systems" that really only includes miniaturized versions of a desktop paradigm. Because of this, the true extent of Jini portability is limited, and suffers fundamental incompatibilities with traditional embedded systems.

As brief examples, the following embedded protocols are either in use or are actively being considered for widespread adoption in embedded applications:

- The Train Control Network uses a combination of fixed-format periodic and aperiodic messages for controlling high-speed trains in two different protocols (one for short-length runs within a vehicle; one for a network spanning an entire train). Messages have 50 to 100 msec deadlines, and run at 1 to 1.5 Mbps. Maximum message payload is 32 bytes. TCP/UDP and IP are not supported. [6]
- TTP (Time Triggered Protocol) is an emerging automotive network protocol that emphasizes determinism and is designed to possibly replace CAN. All messages occur at precisely identified times in a fixed message sequence. Maximum message size of the

first version of TTP/C (for critical networking) was 16 bytes, although newer versions approach 256-byte payloads. TCP/UDP and IP are not supported. [7]

- Some higher-level standards have been able to embrace both Ethernet and embedded protocols. One example of this is BACnet for building automation control. [2] However, this level of standard requires use of specific information formats that do not support off-the-shelf middleware such as Jini, and does not supply the dynamic discovery services needed by RoSES.

5.3. Lessons learned

From this experience and extensive previous experience with other embedded systems and network protocols, we can glean a list of things to keep in mind when designing a system for possible embedded use:

1. Embedded systems tend not to use TCP/IP. Any system that assumes desktop network protocol stacks are available is generally not portable to embedded systems other than high-end desktop-like platforms.

2. Because most control messages have just a few bits of state variables, embedded systems are optimized for short messages, which breaks fragmentation assumptions on desktop networks. It also means headers are skimpy on address bits and node addresses tend to be ambiguous globally — applications have to know which network they are on to identify which physical node has a particular address.

3. Embedded systems do not support sophisticated mechanisms like sockets. They must use the simplest, sparest means possible, to both meet costs and simplify verification and certification for critical applications. Therefore, if something can be adequately accomplished in a single way, embedded systems typically will not support additional mechanisms (especially for communication) that require additional cost and verification.

4. Embedded systems use a variety of network schemes, and tend toward periodic broadcast transmission rather than event-based unicast messaging. Middleware should avoid making assumptions about relative efficiencies of different messaging services.

5. Determinacy and reliability are crucial in many embedded systems no matter the network protocol used. For example, an automotive braking message can't afford to wait indefinitely to find an opening on the network. In this case study we have observed the following issues:

- large message lengths, requiring perfect transmission of large numbers of fragments and re-assembly by the receiver (on embedded networks that don't ensure in-order delivery);
- timely response for crucial actions (for example, with Jini this would include registering with the lookup service and noticing failed nodes); and
- complex sequences of communications that become brittle to message losses that are likely in noisy embedded system operating environments.

The Jini concept needn't have been restricted to a desktop world (and it wasn't intended that such a restriction should occur in practice -- but it did). Jini could have been suitable for real-time embedded environments if desktop-only details had not crept into the "object interface level." This case study suggests that the differences in technology between embedded real-time and desktop computing environments are more significant than many designers appreciate. It also provides some lessons on types of issues to consider with care in creating portable middleware:

1. Hardware is not the only variable in creating portable systems. Software operating environments including operating systems and protocol stacks vary too. TCP/IP is not a universal protocol. RMI is not universally available even on platforms with Java.

2. It is important to justify everything that "leaks" through an abstraction layer. Jini messages seem to have let a bit of *how* they were accomplishing things seep into representations of *what* they were trying to accomplish, especially with respect to multicast/unicast distinctions.

3. Don't over-optimize for today's common case if tomorrow's common case may be different. Optimizing for UDP-sized message fragments and a particular multicast/broadcast tradeoff point complicated attempts to port to another protocol.

During these efforts we corresponded with several Sun employees that were very familiar with Java and Jini. They were surprised by our struggles. Apparently, while touting platform-independence, they had never actually worked out the details of porting to a non-desktop platform and had not encountered the problems that emerge.

6. Conclusion

For the Robust Self-Configuring Embedded Systems (RoSES) project to succeed in its goal of graceful degradation via software reconfiguration, an appropriate run-time infrastructure must be in place to facilitate the "plug-and-play" functionality required for nodes to form a dynamic, ad hoc, distributed network. We investigated various middleware technologies to supply such a run-time infrastructure on top of CAN, ultimately choosing Jini.

In the process of porting Jini to CAN, we discovered that the design of Jini made assumptions about the use of TCP and UDP, including choices of packet sizes and message fields. This did not prevent porting to CAN, but imposed a significant source of inefficiency and difficulty. Unfortunately, higher-level problems lead to very slow reconfiguration times, leading to an overall conclusion that Jini is unsuitable for use on traditional embedded real-time control applications without significant changes.

This case study suggests that the differences in technology between embedded real-time and desktop computing

environments are more significant than many designers appreciate. When evaluating the suitability of an existing OO technology for use in embedded real time systems it is important to question all the underlying assumptions made, especially with respect to network and operating system services. Things to keep in mind when taking something from the desktop world to the embedded world:

- Embedded networks often do not provide all distribution modes (unicast, multicast, broadcast)
- Port numbers, hostnames, and IP addresses are not available in embedded networking
- Message sizes can vary widely, and typically will be significantly smaller than messages destined for a desktop environment
- Determinacy and reliability are not just convenient – they're critical in a real-time world
- Sophisticated mechanisms, especially for communication, are not found in embedded systems; stick to the simplest single way of accomplishing a task

7. 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. We would also like to acknowledge Bill Nace, Keith Thompson, Greg Frazier, Geoffrey Clements, and Mike Bigrigg for their invaluable assistance.

8. 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 *Proc. of the First International Symp. on Object-Oriented Real-Time Distributed Computing,* Kyoto, Apr. 1998, p. 467-76.

[2] Haakenstad, L.K., "The open protocol standard for computerized building systems: BACnet," *Proc. Int. Conf. On Control Applications*, 1999, vol. 2, pp. 1585-1590.

[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 *Proc. of the Sixth IEEE Real-Time Technology and Applications Symp.,* Washington, D.C., June 2000, p. 178-187.

[6] Kirrmann, H.& Zuber, P.A., "The IEC/IEEE train communication network", *IEEE Micro*, vol. 21 #2, March-April 2001, pp. 81-92.

[7] Kopetz, H.; Grunsteidl, G., "TTP - a protocol for fault-tolerant real-time systems," *IEEE Computer*, vol. 27 #1, Jan. 1994, pp. 14-23.

[8] 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.

[9] Nace, W. & Koopman, P., "A Graceful Degradation Framework for Distributed Embedded Systems," *Workshop on Reliability in Embedded Systems*, October 28, 2001 (in press).

[10] 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.

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

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

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

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

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

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

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

[18] Waldo, J. "The End of Protocols", Sun Microsystems *Java Developer Connection*, June 2000.

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