Flexible Fault Tolerance in Configurable Middleware for Embedded Systems

Kevin Dorow  
Pacific Northwest National Laboratory  
Richland, WA 99352 USA  
Kevin.Dorow@pnl.gov

Abstract

MicroQoS CORBA (MQC) is a middleware platform that focuses on embedded applications by providing a very fine level of configurability of its internal orthogonal components. Using this configurability, a developer can generate a customized middleware instantiation that is tailored to both the requirements and constraints of a specific embedded application and the embedded hardware.  

One of the key components provided by MQC is a set of fault-tolerant mechanisms, which allow for support of applications that require a higher level of reliability. This document provides a detailed description of the algorithms and protocols selected for these mechanisms, along with a discussion of their implementation and incorporation into the MQC platform.

1. Introduction

Advancements in computer technology over the past decade have created an explosion in the number of embedded systems. The number of devices manufactured across this spectrum is staggering, at eleven billion parts per year. And soon, as ubiquitous networking becomes more prevalent, each of these systems will have the potential for connecting and interacting with one another to, hopefully, provide a better and more integrated level of service.

This explosion obviously opens up a large amount of opportunities for the computing industry. However, there are drawbacks associated with these opportunities. Development on embedded systems is typically fraught with difficulties. Because of economical constraints, for example, the available resources of embedded system devices are extremely restricted, not only with regard to storage and computing capacity, but also to bandwidth and reliability of communication connections. Also, embedded systems tend to rely on proprietary, low-level mechanisms and interfaces that are designed to function in a specific application. In addition, due to the rapid evolution of computing technology, the components associated with an embedded system can quickly become obsolete and unsupported (sometimes even before a system is released into a production environment).

In order to try and overcome the difficulties associated with distributed (but not specifically embedded) systems, middleware technologies have been developed and matured. Middleware is a class of software technologies designed to help manage the complexity and heterogeneity inherent in distributed systems [1]. It significantly reduces the burden on application programmers by providing them with high level, platform independent abstractions that access low level platform specific functions.

Current middleware frameworks are being applied to small, embedded systems with varying degrees of success. Many high-end workstations have more onboard CPU cache than some embedded systems have available memory in both RAM and ROM. Thus, general-purpose frameworks developed for these memory rich environments often fail to scale down to memory-starved environments. Other solutions have been developed for small, embedded devices, but typically are point solutions, supporting one hard-coded set of constraints and tradeoffs. Consequently, they do not have the flexibility to cover the wide space of the embedded systems market, or in many cases to provide the particular embedded application designer with the right set of constraints appropriate for the specific application functionality and target hardware.

Recent specifications such as the minimumCORBA (Common Object Request Broker Architecture) and Real-Time CORBA specifications have begun to address some of the concerns associated with the embedded systems market, but not to the extent required [3].

Besides addressing the issues associated with embedded systems, existing middleware solutions such as CORBA also tend to overlook the two most fundamental problems which occur in real-world distributed applications, namely partial failures and consistent ordering of distributed events [5].

Taking all these issues into account, we have designed and implemented MQC, a configurable middleware framework for embedded systems. It allows the middleware to be tailored both to the precise properties of the hardware as well as to the application’s configuration requirements. We realized during its
design and development that small, embedded device middleware has many facets associated with it. Consequently, MQC has been designed to address many of these facets, such as fault tolerance, security, power usage, and system performance.

Altogether, the MQC provides a very powerful middleware framework that is:

- targeted specifically for embedded distributed systems
- is highly configurable (which is required by the large swing in requirements by different embedded system applications)
- provides toolkits so that even beginning embedded system programmers can take advantage of the framework's flexibility.

The focus of the work described in this paper is to integrate fault-tolerance into the MQC framework in a configurable way. Consequently, the contributions of this work are:

- the analysis, design, and implementation of several fault-tolerant mechanisms into the MQC framework
- the analysis, design, and implementation of a highly configurable fault-tolerant group communications system

The rest of this document is organized as follows: Section 2 explains the fault-tolerant properties that were identified as candidates to be integrated into MQC. Section 3 describes the implementation details associated with each of the fault-tolerant properties listed in section 2. Section 4 presents the results of the implementation and associated analysis. Section 5 contains the conclusion.

### 2. Configurable Fault-Tolerant Mechanisms

Most applications in the realm of embedded systems require some level of dependability in order to be successful. In this context, dependability can be defined as follows: "The measure in which reliance can justifiably be placed on the service delivered by a system" [6]. The need for this dependability stems from the fact that many embedded systems are mission critical (i.e. fly-by-wire systems in airplanes, anti-lock braking systems for cars, remote banking systems, etc.). It is because of this need of dependability that we felt it was important to add fault-tolerant mechanisms to MQC.

The first step in adding fault tolerance into MQC was to examine the space defined by embedded system applications and identify fault-tolerant mechanisms that could be beneficial. A review of each identified mechanisms was then done to determine the feasibility of creating an implementation in the MQC architecture.

Table 2.1 summarizes the fault tolerant mechanisms that were selected to be incorporated into the MQC framework

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Reliability</th>
<th>Ordering</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spatial</td>
<td>Group Communication</td>
<td>Consensus</td>
</tr>
<tr>
<td>Value</td>
<td>FEC</td>
<td>Leader Election</td>
</tr>
<tr>
<td>Temporal</td>
<td>Load Balancing</td>
<td>Majority Voting</td>
</tr>
</tbody>
</table>

Table 2.1 Fault Tolerant Mechanisms in MQC

These mechanisms will be discussed in more detail in the following subsections.

#### Temporal Redundancy

Temporal redundancy consists of doing the same thing more than once, in the same or in different ways, until the desired effect is achieved. A simple example of temporal redundancy is the retransmission of a message in order to tolerate omissions due to electromagnetic noise or temporary receiver overflow [7].

#### Spatial Redundancy

Spatial redundancy consists of having several copies of the same component. The same information can be stored in several disks, tolerating the loss of one disk (if disks are placed very far apart, we can even tolerate events such as floods or fires). Different nodes can compute the same result in parallel to ensure that, even when one of them crashes, the result is available on time (active replication). In a distributed system, information can be disseminated along different network paths, to survive physical media damage [7].

#### Value Redundancy

Value redundancy consists of adding extra information about the value of the data being stored or sent. This extra information is normally control data in the form of codes that allow the detection, or even the correction, of integrity errors in the data being stored or transmitted. For instance, a parity bit or an error correcting code can be added to memory chips or to disk structures, respectively to detect or detect / correct data corruption. Frame check sequences or cyclic redundancy checks can be added to the data being transmitted in order to detect multi-bit corruption by noise.

#### Reliability

Reliability is a fault-tolerant property with two underlying mechanisms that have been selected for implementation in MQC. These mechanisms are group communication and failure detection.

#### Group Communication

According to [4], group communication is a means for providing multi-point to multi-point communication, by organizing processes in groups. A group is a set of processes which are members of the group. Each group is associated with a name. Processes
communicate with the group members by sending a message targeted to the group name; the group communication service delivers the message to the group members.

Group communication systems have two main aspects associated with them: reliability aspects and ordering aspects [7]. Reliability aspects deal with the message delivery guarantees. Ordering aspects deal with the message ordering guarantees. As we will see in later sections of the paper, the group communication system that has been implemented in the MQC framework supports multiple levels of both reliability and ordering to provide even more flexibility.

Group communication systems are a fundamental building block of fault tolerant distributed systems. Typical applications in which group communication systems are used include all forms of replication (active, primary-backup, etc.), support for distributed and clustered operating systems, distributed transactions and database replication, load balancing, and collaborative computing.

Failure Detection
Failure detection is any mechanism that can detect and diagnose failed components in a system. It is fundamental to fault-tolerance in that a very important aspect of being able to handle errors in a system involves being able to identify where they are coming from. Once this is accomplished, the source can be either disconnected or repaired—either way resources are no longer wasted trying use the component, which also helps in the area of performance.

3. Implementation and Integration

This section will present the implementation and integration details of each of the fault-tolerant mechanisms described in the previous section. In addition, future areas of follow-on work will be presented.

Temporal Redundancy
Temporal redundancy for the communication channel was chosen as the specific implementation of the temporal redundancy mechanism in MQC. This type of redundancy allows the system to tolerate up to \( k \) omission failures by allowing the user to specify a fixed number of automatic retransmissions \((k+1)\). The failure model that is supported by this mechanism is omissive transient faults.

This mechanism is configurable from the configuration tools (a.k.a. CASE tools) that are used to generate a custom ORB that fits the QoS requirements for a particular application. The user is able to specify the number of omission failures that need to be tolerated in the CASE graphical user interface (GUI). Based on this selection, the code for both the client and server portions is generated with the following functionality: (1) The message sending mechanism for both the client and server, which resides in the Transport class files, is modified to send the required number of retransmissions of every message delivered. (2) The message receiving mechanism for both the client and server, which resides in the Delegate class files for the client and the Handler class files for the server, is modified so that duplicate messages are not processed. Specifically, unique messages are identified by an identification (ID) field in the header of the message. When a message is received, its ID field value is extracted and then compared against a message history buffer that stores the ID field values of the last 50 messages that have been received. If a match is found, the message is assumed to be a duplicate and is not processed further. If a match isn’t found, the message is assumed to be a new message—the ID field value is loaded into the message history buffer and the message is passed on to the ORB for further processing.

An area of follow-on work would be to provide some options for how to process the duplicate transmissions that are received. For example, in order to provide a higher quality of service, one may elect to compare the duplicate messages to identify possible errors in transmission. Above and beyond error detection, it may be beneficial to support a voting on the agreed upon message content by a majority of the retransmissions (in the presence of errors).

Spatial Redundancy
Spatial redundancy for the communications path was chosen as the specific implementation of the spatial redundancy mechanism in MQC. This type of redundancy allows the user to specify multiple communication paths (i.e. Ethernet, one-wire) to deliver messages for message omission and channel failure tolerance. The failure model supported by this mechanism is omissive transient and permanent faults.

This mechanism is also configurable from the CASE tools—the user can specify multiple communication paths to be supported by both the client and server. The generated client and server codes are then modified to support the selected paths. On the client, the Delegate class is modified to contain a collection of transports (which is populated with the transports that correspond to the selections made in the CASE tools). When message sending and receiving takes place, it is processed by all transport elements in the collection. On the server, multiple message handlers are instantiated for the communication types selected from the CASE tools. This allows the server to process messages from all different communication paths that need to be supported.

Areas for follow-on work include the following: (1) The client ignores any duplicate messages that are
received from other communication paths. It may provide value (just as with the temporal redundancy) to provide some options for how the duplicate messages are processed. These options could range from simple error detection (detection of differences in any of the messages) to voting to provide the value agreed upon by majority of the duplicate transmissions (depending on the number of redundant channels). (2) The server processes all messages that are received (even the duplicate messages on the redundant communication channels). This may not be suitable for a server application that can update its state based on messages received. A way to overcome this may be to implement a mechanism to detect the duplicate messages that are received and be able to respond to them without processing them (by sending the same response that was sent for the original message).

**Value Redundancy**

Value redundancy in the form of a checksum included in messages was chosen as the specific implementation of the value redundancy mechanism for MQC. This type of redundancy allows the contents of a message to be validated before being processed. The failure model supported by this mechanism is assertive transient faults.

This mechanism is selectable from the CASE tools GUI. When selected, the transport classes of both the client and server are modified so that a checksum value is calculated and included with each message that is sent. Upon receipt the checksum value is extracted and a checksum is calculated on the rest of the message to validate the contents. If the computed value does not match included value, the message is rejected (since it is corrupted).

Areas of follow-on work include incorporating more sophisticated error correction codes so that a corrupted message could be repaired instead of dropped. Another option would be to incorporate code to request retransmissions of messages that were detected as corrupted.

**Reliability**

As mentioned in section 2, reliability is implemented in the forms of group communication and failure detection mechanisms for MQC. The following subsections provide the implementation details for each.

**Group Communication**

In order to provide value to some related projects, it was determined that a stand-alone group communications system would be created and then integrated into MQC. The resulting mechanism is highly configurable and supports the following types of group communication (note—all algorithms are based on those found in [2]):

- **Non-uniform Failure-Atomic Multicast**—In this mode, the message sender adds the membership list for the group at the time of the send to the header of the message. The sender then sends point-to-point messages to all in the list. Upon receipt the receiver delivers the message to the application and then stores the message in a buffer until told that it is okay to delete. It also sends an acknowledgement (ACK) back to the sender to acknowledge the fact that it received the message. Meanwhile, the sender waits to receive ACKS from all the members on the list. Once all the ACKS are received the sender sends a message to all the members telling them to delete the message copy (still store the message ID, though, to be able to handle rejecting duplicates). Upon receipt of this message the members do as they are told and delete the message copy. The message ID is stored in a fixed length buffer, so eventually it will be purged when enough future messages are received.

- **Dynamically Uniform Failure-Atomic Multicast**—In this mode, the same procedure described for Non-uniform Failure-Atomic Multicast is used, but another round of messages is added before the first one. In this round the message is sent and the receiver must buffer (and not deliver it) until the sender has received ACKS from all members and then returned the “ok to deliver” message.

- **FIFO Ordered Multicast**—In this mode, FIFO (First-In-First-Out) ordering of messages is implemented on top of the two Failure-Atomic multicast protocols listed above by adding an ordered message id for each sender (a counter) to the message header. When messages are received they are delivered to the application in an order that matches the sequence of message ids (if a message arrives out of sequence, it is held in a buffer until it is next in the sequence to be delivered). This process results in fbcast, which is non-uniform multicast with sender-based FIFO ordering, and safe fbcast, which is dynamically uniform multicast with sender FIFO ordering.

- **Causal Ordered Multicast**—In this mode, the implementation relies on a Vector Timestamp protocol to create the ordering (on top of the two failure atomic protocols to provide reliability). The Vector Timestamp is a count of causally prior messages for each process. A sender labels a message with the current Vector Timestamp at its host. The receiver of the message delays delivering it until (1) it is the next message in the sequence from the sender, and (2) every causally prior message from the other processes has been delivered.

- **Totally Ordered Multicast**—In this mode, a moving sequencer algorithm was implemented to supply the total ordering (ignoring the causality requirement). In the sequencer implementation, a single member of the group is responsible for supplying the ordering of the messages (the messages are sent to all members and cached until the sequencer sends the ordering for that message). The “moving” part provides fault tolerance in
that if the sequencer fails, another group member takes over the sequencer role.

All of the above modes for the group communication system are selectable from the CASE tools GUI. The group communication package is then integrated as a member of the server orb class. Also, the server transport classes are modified to multicast any messages received to all group members. This allows for replicated servers to be created and maintained.

There are several areas for follow-on work. The group communication package could be expanded to support more communication modes (virtual synchrony, total ordering with causality, etc.). Also, options could be provided to specify how the responses from replicated servers should be handled (could range from error detection to voting and majority approved values). Group communication support among clients could also be explored.

**Failure Detection**

Failure detection has been implemented in the context of the group communication system for MQC. The user can specify a timeout period for messages as part of the group communication system configuration in the CASE tools GUI. Using the value, the group communications package has code to monitor both the senders and receivers of messages. If responses to messages do not occur within the specified timeout period, the associated group member(s) is removed from the group and a view change message is multicast to the remaining group members. Any messages that were in transit when failures occur are handled via the failure atomic protocol selected (if the message was partially delivered to the group, the one of the receiving group members will assume the responsibility of finishing the message stream).

### 4. Results and Analysis

In this section we will examine the impact of the fault tolerant property implementations on the performance of MQC. In particular we will focus on the impacts on memory footprint and time of execution.

**Test Platform**

The testing described in the following subsections was executed on three test platforms. The first test platform was a desktop system with an Intel Pentium 4, 1.5 GHz processor, 256 MB RAM, running Linux (kernel version 2.4.9). The second test platform was a Systronix Saje board, which has a 100 MHz aJile Systems aJ-100 CPU that executes Java byte-code natively. The third platform was a Dallas Semiconductor TINI board, which is powered by a ~40 MHz DS80C390 CPU.

**Impact on Memory Footprint**

The “simple” application located in the MQC sample directory was chosen for memory footprint analysis. Below are two tables: The first contains the total size of the class files associated with the server with the various fault tolerant properties selected.

<table>
<thead>
<tr>
<th>Application: Simple -- (use TCP and UDP unless stated otherwise)</th>
<th>Server Class File Size (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Linux</td>
</tr>
<tr>
<td>No Fault Tolerance</td>
<td>85,585</td>
</tr>
<tr>
<td>Partial Redundancy</td>
<td>85,360</td>
</tr>
<tr>
<td>Spatial Redundancy (TCP and Unreliable UDP)</td>
<td>82,346</td>
</tr>
<tr>
<td>Group Consensus (TCP)</td>
<td>79,900</td>
</tr>
<tr>
<td>Group Consensus (UDP)</td>
<td>192,960</td>
</tr>
<tr>
<td>Value Redundancy</td>
<td>79,900</td>
</tr>
</tbody>
</table>

**Table 4.1 Memory Footprint of Server Files**

Notice that the impact to the server files for the properties other than the ones associated with Group Communication is almost negligible (the largest is spatial redundancy with ~20 kilobytes—this is due to the added support for the additional Unreliable UDP transport). The impact of the options associated with Group Communication is significant, though (over double the class file size for dynamically uniform atomic multicast support). This is mostly due to the fact that the Group Communication package was developed as a stand alone module, and consequently, has its own transport code incorporated into it. The memory footprint associated with these modules could be significantly reduced by taking advantage of the already existing transport mechanism in MQC.

The next table contains the total size of the class files associated with the client with various fault tolerant properties selected.

<table>
<thead>
<tr>
<th>Application: Simple -- (use TCP and UDP unless stated otherwise)</th>
<th>Client Class File Size (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Linux</td>
</tr>
<tr>
<td>No Fault Tolerance</td>
<td>81,126</td>
</tr>
<tr>
<td>Partial Redundancy</td>
<td>78,910</td>
</tr>
<tr>
<td>Spatial Redundancy (TCP and Unreliable UDP)</td>
<td>77,895</td>
</tr>
<tr>
<td>Group Consensus (UDP)</td>
<td>75,450</td>
</tr>
<tr>
<td>Value Redundancy</td>
<td>192,960</td>
</tr>
</tbody>
</table>

**Table 4.2 Memory Footprint of Client Files**

Notice on the client that the impact to the class file size is negligible (~1 kilobyte or less) except in the case of Spatial Redundancy, where the extra 10 kilobytes is the code needed to support the secondary Unreliable UDP transport.

**Impact on Execution Times**

The “timing” application located in the MQC sample directory was chosen for execution time analysis. It is a simple application that contains one method in which a long value is passed from the client to the server and back again. Inside the client the call to the method is made iteratively and timing measurements are captured.
during each iteration. The average call time is then calculated and presented. In the subsections that follow, the data acquired for this application with different fault tolerant properties selected will be presented and discussed.

Temporal Redundancy

Below is a table showing the comparison of timing results with no fault tolerant properties selected and varying levels of temporal redundancy on the three testing platforms.

<table>
<thead>
<tr>
<th>Application Timing (Use TCP and UDP unless stated otherwise)</th>
<th>Linux</th>
<th>TINI</th>
<th>AJILE</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Fault Tolerance</td>
<td>387</td>
<td>273</td>
<td>375</td>
</tr>
<tr>
<td>Temporal Redundancy (1 copy)</td>
<td>397</td>
<td>283</td>
<td>384</td>
</tr>
<tr>
<td>Temporal Redundancy (5 copies)</td>
<td>576</td>
<td>672</td>
<td>576</td>
</tr>
<tr>
<td>Temporal Redundancy (7 copies)</td>
<td>576</td>
<td>672</td>
<td>576</td>
</tr>
</tbody>
</table>

*Table 4.3 Execution Times with Temporal Redundancy*

From the data it can be determined that the impact associated with temporal redundancy is fairly linear for resource rich processing platforms—on the Linux desktop platform there is a 30 microsecond overhead (approximate) associated with each additional retransmission. This overhead is associated with the additional processing required to send the duplicate transmissions along with filtering out any duplicate transmissions that are received. As the processing horsepower of the platform drops down, the impact of this overhead becomes much more significant, as can be seen by the AJILE and TINI results. It is also evident from the AJILE results that there are some issues on this platform that have yet to be resolved.

Another important finding in the testing was that it is critical to consider the message sending rate when selecting the amount of temporal redundancy to support. The timing application that we used initially did not limit the message sending rate—this caused message packet corruption problems with the higher levels of temporal redundancy. An area of future work may be to modify the transport code so that it can be configured to throttle the message sending rate to a given value (which could be set in the CASE tools).

Spatial Redundancy

The table below shows the execution times with no fault tolerant properties selected and with spatial redundancy (using TCP and UDP transports).

<table>
<thead>
<tr>
<th>Application Timing (Use TCP and UDP unless stated otherwise)</th>
<th>Linux</th>
<th>TINI</th>
<th>AJILE</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Fault Tolerance</td>
<td>387</td>
<td>273</td>
<td>375</td>
</tr>
<tr>
<td>Spatial Redundancy (TCP and UDP )</td>
<td>294</td>
<td>194</td>
<td>NA</td>
</tr>
</tbody>
</table>

*Table 4.4 Execution Times with Spatial Redundancy*

From the table data it can be seen that spatial redundancy causes a doubling of the execution time. This is caused by the addition of code to filter out and eliminate processing of duplicate messages along with overhead associated with sending a message over an additional transport mechanism. Testing results were not available for the TINI and AJILE platforms due to a lack of support for the current UDP implementation in MQC.

Value Redundancy

The table below shows the execution time results when running the “timing” application with no fault tolerant properties selected and with value redundancy enabled.

<table>
<thead>
<tr>
<th>Application Timing (Use TCP and UDP unless stated otherwise)</th>
<th>Linux</th>
<th>TINI</th>
<th>AJILE</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Fault Tolerance</td>
<td>387</td>
<td>273</td>
<td>375</td>
</tr>
<tr>
<td>Value Redundancy</td>
<td>294</td>
<td>194</td>
<td>NA</td>
</tr>
</tbody>
</table>

*Table 4.5 Execution Times with Value Redundancy*

The data in the table shows a slight increase in the execution time with value redundancy enabled on the resource rich platforms. This increase is due to the computing time required to calculate the CRC32 checksum on both the sending and receiving sides of a message delivery, along with the incorporation and extraction of the checksum from the message body. It is evident from the TINI and AJILE results that computing the checksum is processor intensive, hence the much more dramatic increase in average call time.

Group Communication

In order to test the execution time impact of group communication, the server code of the “timing” application was modified to support active replication. This means that whenever any CORBA message was received by a member of the server group, it used the group communications package to distribute the same message to all other group members. The table below shows the execution time results when running the modified “timing” application with no fault tolerant properties selected and with various group communication options selected (note group communication testing was not performed on the AJILE and TINI boards because of some issues in converting code to function properly on these platforms).
Table 4.6 Execution Times with Group Communication

<table>
<thead>
<tr>
<th>Application Timing (Use TCP and ORP unless stated otherwise)</th>
<th>Execution Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Fault Tolerance</td>
<td>367</td>
</tr>
<tr>
<td>Group Comm (Non-Uniform-3 members no order)</td>
<td>680</td>
</tr>
<tr>
<td>Group Comm (Non-Uniform-3 members total order)</td>
<td>752</td>
</tr>
<tr>
<td>Group Comm (Non-Uniform-3 members no order)</td>
<td>360</td>
</tr>
<tr>
<td>Group Comm (Non-Uniform-4 members no order)</td>
<td>373</td>
</tr>
<tr>
<td>Group Comm (Non-Uniform-4 members total order)</td>
<td>1253</td>
</tr>
<tr>
<td>Group Comm (Uniform-2 members no order)</td>
<td>1120</td>
</tr>
<tr>
<td>Group Comm (Uniform-2 members total order)</td>
<td>764</td>
</tr>
<tr>
<td>Group Comm (Uniform-3 members no order)</td>
<td>231</td>
</tr>
<tr>
<td>Group Comm (Uniform-3 members total order)</td>
<td>3000</td>
</tr>
<tr>
<td>Group Comm (Uniform-4 members no order)</td>
<td>1591</td>
</tr>
<tr>
<td>Group Comm (Uniform-4 members total order)</td>
<td>1790</td>
</tr>
</tbody>
</table>

From the data in the table it can be seen that the impact to the execution time is significant (from ~3 times longer to ~5 times longer). This was expected for several reasons. As mentioned previously, the group communication package was developed as a stand alone module. Consequently, it uses its own encoding and transport mechanisms. Large gains in performance could be achieved by modifying the package to take advantage of communication mechanisms that already exist in MQC (since they are already optimized and configurable). Another factor that contributed to the time increase is due to the code necessary to generate the group communication message format (which increases in complexity and length based on the ordering mechanism selected). The increase is also due, in part, to the fact that group communication requires the sending of multiple messages to all group members (sending more messages takes more time).

5. Conclusion

Summary

MQC is a middleware framework targeting small, embedded systems. MQC is small, highly configurable to a fine level of granularity, offers support for multi-property Quality of Service and has the ability to scale down to memory-starved environments. In particular, MQC supports a fundamental set of QoS properties related to fault tolerance, which allows it to address the problems commonly found in real-world distributed applications—partial failures and consistent ordering of distributed events. Together with a development toolkit, MQC architecture offers a development environment for embedded systems developers.

The contribution of this work was to help develop MQC, and in particular to integrate a set of configurable fault-tolerant mechanisms so that applications requiring a higher level of reliability could be supported. In order to accomplish this task, research was first done to determine a set of target fault-tolerant mechanisms that would be useful in typical embedded system applications. A design phase followed to verify the feasibility of integrating the selected mechanisms into the MQC framework in an orthogonal way. Once the feasibility and design details had been determined, the selected mechanisms were implemented and incorporated. Testing and benchmarking of the new mechanisms was then performed on selected platforms. An analysis of the results along with directions for future work were identified and documented.

6. References


