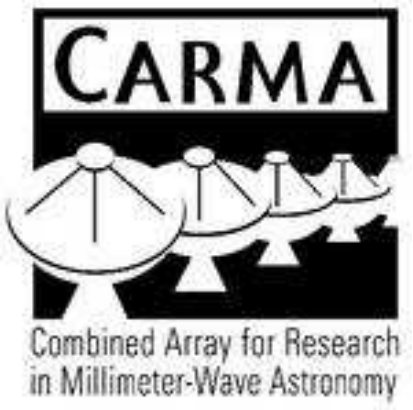

CARMA Memorandum Series #22

ORBacus Notify Performance in CARMA

Chul Gwon, N. S. Amarnath
University of Maryland
Tom Costa
OVRO

April 5, 2004

ABSTRACT

The CARMA Monitor System (CAM) (1) was designed to allow efficient sharing of data between distributed subsystems within CARMA. It was proposed that the transfer of this data between the subsystems be accomplished using ORBacus Notify (2). To insure that this would be a viable option given the CPU constraints of CAM, we conducted performance tests on Notify. This document discusses these performance tests, as well as the eventual decision to use ORBacus Notify.

1. Introduction

The Notification Service is an Object Management Group (OMG) specification (3); its purpose is to decouple communication between CORBA objects. Rather than objects communicating directly with each other as servers and clients, they communicate through an intermediate object (an *event channel*) as suppliers and consumers. Using an event channel allows multiple suppliers to transmit data to multiple consumers without the suppliers and consumers knowing about each other. The Notification Service also includes the ability to set Quality-of-Service (QoS) parameters and filter events for different consumers. ORBacus Notify is an implementation of this OMG specification.

In the framework of the CARMA Monitoring System (CAM), suppliers would consist of the various CARMA subsystems from which monitor points would be transmitted (antennas, correlators, etc). The Event Channel would receive these monitor points and send them to a single consumer residing on the ACC. Since the ACC will be responsible for more than just the collection of monitor points, a "CPU budget" was determined to specify the allowed amount of CPU usage for any single process. It was therefore necessary to conduct performance tests on ORBacus Notify to insure that it fell within the limits of the allowed CPU usage. This document discusses our performance tests as well as the ORBacus Notify configuration options that we used for our studies.

2. Notification Service Configuration

The configuration options discussed in this section involve only the ones that we found pertinent for the CAM. A full list of options are available in the IONA documentation (2). The options here are divided into those whose values need to be changed from the defaults and those that should retain their defaults.

2.1. Options with values different from defaults

- *ooc.database.max_transactions*: The maximum number of allowed active concurrent transactions. This value should be proportional to the total number of persistent proxies. The default is 20; since we have 50 suppliers and 1 consumer, we set our value to 102.
- *ooc.database.max_locks*: The maximum number of database locks that can be acquired at any time. The default is 16384. For some reason, when initially experimenting with the notification service and 50 suppliers, the error "Database: Unable to allocate memory for transaction detail" arose. By increasing the maximum number of locks, this error disappeared. The value we used was 262144.

2.2. Options using default values

- *ooc.database.max_retries*:

The maximum number of times the notification service will retry a transaction before aborting. It is quite common for the service to attempt several retries for any given transaction, sleeping a few milliseconds between retries (the sleep grows exponentially, but events are typically sent significantly before the half-second tick). Setting this value to anything but its default (0, meaning unlimited retries) could result in a significant loss of events. Tests showed that setting this value to 1 resulted in a 20-50% loss of events.

- *ooc.notification.eventqueue*:

  This option is supposed to further isolate the consumers and suppliers by using a single event queue rather than multiple queues. Using this option results in an increased the number of transactions, which also increases the latency time for events by a factor of two. Therefore, for purposes of speed, we keep this set to the default: false.

- *ooc.notification.dispatch_strategy*: This option only affects threads that the notification service uses for event delivery. For example, when using a pull supplier, the service must initiate the event delivery from the supplier. For CAM, a push supplier and pull consumer are used, so the channel never has to initiate the delivery. This option therefore has no effect on CAM.

3. QoS parameters

A benefit of using the notification service over the event service is that it supports quality of service (QoS) parameters for controlling event queuing and lifetime; specifically, things including event expiry times, connection persistence, and filtering options. For CAM, most of these options are applied to the channel; the admins and proxies from the channel then inherit these values where applicable.

- *ConnectionReliability*:

  This refers to the ability of the notification service to restore all object connections after it is restarted. This option must be set to Persistent when used in conjunction with the Implementation Repository (IMR) and object activation daemon (OAD)[1]. Consumers/Suppliers will receive a COMM_FAILURE if they are actually *in the process* of receiving events when the Notification server dies, otherwise they will not receive any error. Initial tests indicate that if Notification server dies while attempting to transfer an event, it will resend that event upon being restarted.

- *EventReliability*: If set to Persistent, this would guarantee that consumers will get all events obtained by the notification service from suppliers (barring any filtering options). Unfortunately, we have run into a various number of problems with this option, such as unusually long hangs and overwhelmingly

[1]The IMR provides indirect bindings for persistent object references; therefore, clients communicate through the IMR to locate and connect to a desired server. Through the OAD, the IMR can (re-)start servers on demand if the server is not active

large DB files. Our tests have shown that even with this option set to BestEffort, all events still get through without the issues discussed above. Therefore, this option is left defaulted to BestEffort.

- *OrderPolicy*: This is the order that events are queued for delivery to a consumer. The default for this is PriorityOrder, which means that they will be based on a priority level that can be optionally set for events. We opted to change this to FifoOrder, since there are no priority levels set for the events and since FIFO makes the most sense for the CAM design.

- *MaxEventsPerConsumer*: Limits the number of events that will be queued in a proxy supplier. This should not be set since setting a limit may result in a loss of events.

- *DiscardPolicy*: This option only takes effect if a queue reaches the limit specified by the MaxEventsPerConsumer property (which should not be set to begin with).

- *Timeout*: By default, events do not have an expiry time. If this value is not set, the CPU and memory usage for a processor increases because the events must continually be dealt with. Ideally, since events are sent on the half-second, we should not need the event after this. However, given that it takes some time to process the event in the Notification server, we have set this value to 1 second.

4. Performance Results

4.1. Mock Monitor System

We created a mock monitor system consisting of 50 suppliers, each pushing 34 KB of data (not including the overhead for using a Structured Event or timing information) at 2 Hz. Three machines were employed for our tests: a 2.8 GHz Pentium-4 desktop, a 1.6 GHz dual-Athlon (rated at a 2.0 GHz Pentium-4), and a 2.0 GHz Mobile Pentium-4 laptop. Unless otherwise specified, all suppliers ran on the laptop, the consumer(s) ran on the 1.6 GHz Athlon and the Notification server ran on the 2.8 GHz P-4.

- **suppliers staggered/unstaggered**: A check was done to see if there is a difference on the performance of the Notification Service between all 50 suppliers sending their events at the same time (unstaggered) and having them send events in 5 ms intervals (staggered). The tests showed that there was not a significant difference in the CPU load. The latency[2] is higher for staggering, however, since the 5 ms wait between each of the 50 suppliers introduces an additional 0.25 s of latency. We can modify the 0.25 s by changing the 5 ms staggering, but this would most likely change the latency caused by the Notification Service (0.1 s) as well.

[2] *latency* for our purpose is the measure of time between sending a subsystem frame from a supplier and receiving it in a consumer.

Test	Consumer CPU (%)	NS CPU (%)	latency (s)
staggered	23	28	0.1 + 0.25
unstaggered	24	26	0.2

Table 1: Staggered vs. Unstaggered Suppliers: CPU usage for the Consumer and Notification server (NS) with latency times for events. The 0.25 s is separated since this is the result of staggering the 50 suppliers by 5 ms and not a result of the Notification Service.

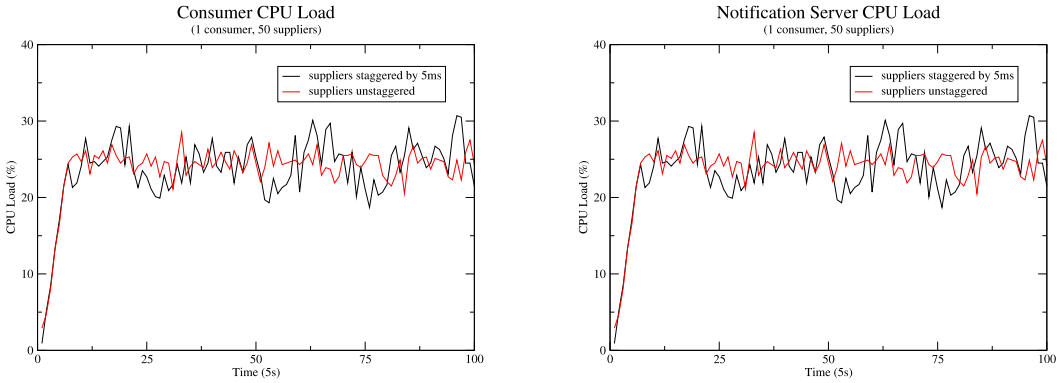

Fig. 1.— Monitor System Consumer and Notification Server CPU usage with staggered and unstaggered suppliers.

- **multiple consumers**: Although the design currently does not call for multiple consumers on a channel, this is an important feature of the Notification Service that may be exploited later. This test shows the effect of adding additional consumers to a channel.

Test	CPU (%) per consumer	NS CPU (%)	latency (s)
1 consumer	23	28	0.1 + 0.25
2 consumers	25	41	0.1 + 0.25
3 consumers	25	57	0.13 + 0.25

Table 2: Using multiple consumers on a single event channel

- **event size effects on CPU usage**: It is believed that changes in the size of events passed through the channel are directly proportional to the CPU usage. To confirm this, we passed events that were 23 KB in size (rather than 34 KB) and measured CPU usage. The result of this indicates that the CPU usage of the consumer is proportional to the event size while the Notification server CPU usage and

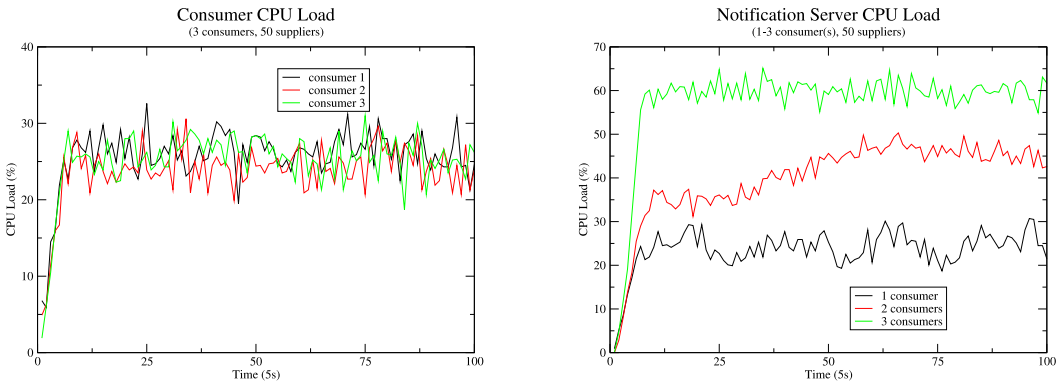

Fig. 2.— Monitor System Consumer and Notification Server CPU usage with multiple consumers. The consumers shown are the CPU usage when three consumers are connected. The Notification server plot shows the CPU usage when each additional consumer is added.

latency time are not.

Event Size	Consumer CPU (%)	NS CPU (%)	latency (s)
34 KB	23	28	0.1 + 0.25
23 KB	16	12	0.03 + 0.25

Table 3: Varying size of events to see effects of CPU usage. If CPU scaled directly with size, we would expect to see 16% for the Consumer and 19% for the Notification server.

4.2. Mock Pipeline System

It was assumed that ORBacus Notify would be used for the Correlator Pipeline system as well as the Monitoring system. The Pipeline system would consist of 16 suppliers, each pushing 13 KB of data at 2 Hz. We measured the CPU loads for the Pipeline System running alone, as well as alongside the Monitor system. The machines used were the same as with the Mock Monitor System: all consumers running on a 1.6 GHz dual-Athlon machine, the Notification server on a 2.8 GHz P-4, and suppliers all running on a 2.0 GHz Mobile P-4.

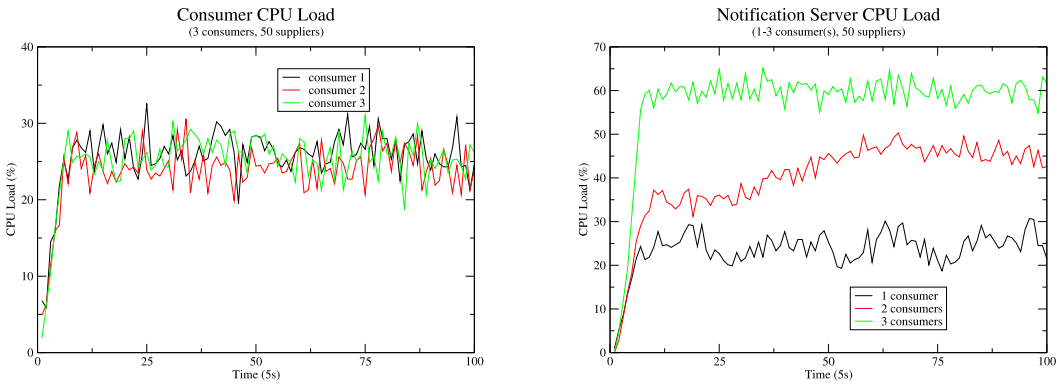

Fig. 3.— Monitor System Consumer and Notification Server CPU usage with multiple consumers. The consumers shown are the CPU usage when three consumers are connected. The Notification server plot shows the CPU usage when each additional consumer is added.

Test	Pipeline CPU (%)	Monitor CPU (%)	NS CPU (%)
Pipeline Only (float [ ])	2	N/A	1
Pipeline Only (monitorframe)	3	N/A	7
Pipeline (float [ ])+ Monitor	2	22	25

Table 4: Mock Pipeline and Monitoring system. *double [ ]* and *monitorframe* refer to the structure that was used to pass data through the event channel.

4.3. Unoptimizied vs. Optimized ORBacus

Studies showed that using the optimization flag "-O3" improves performance of ORBacus objects by 20%; however, the Notification server does not show any improvement in performance. Another important observation is that there was a degradation in performance when ORBacus libraries were built with optimization and the CARMA libraries were not.

4.4. CORBA push model

Independent tests were conducted which were not based on the Notification Service, but rather used a straight CORBA client-server connection and pure socket calls with no CORBA. Paralleling with our benchmarks, there were 50 pushers sending data at 2 Hz to a single catcher. The data size used was 45 KB (we had also initially used 45 KB packets with our tests, but eventually had to decrease to 34 KB packets because of network driver problems (Sec. 5). This study also compared the CPU usage for transferring the same data in various forms as well as using a regular structure instead of a CORBA::Any, which is a data type that

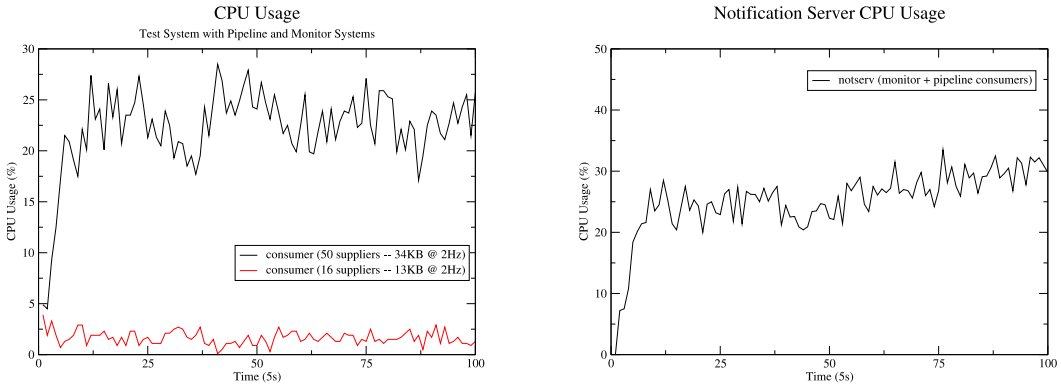

Fig. 4.— Consumer and Notification Server CPU usage for the mock Pipeline and Monitor systems. The consumers shown are the CPU usage when three consumers are connected. The Notification server plot shows the CPU usage when each additional consumer is added.

can be used to represent any possible IDL data type. All processes were run on a single 1.6 GHz Pentium-4 desktop.

The result of these tests showed that the transport structure defined in monitorframe.idl was contributing to the increased CPU usage. Also that there is an additional hit taken when marshalling/unmarshalling a union-type and a struct into a CORBA::Any. The study with the sockets give us a baseline measurement to see the fastest data transport possible, showing the CPU usage introduced by using CORBA. We discuss comparisons with ORBacus Notify in Sec. 4.5.

Test	Total CPU (%) for struct	Total CPU (%) for CORBA::Any
sockets	3	N/A
monitorframe.idl	11.5	28
faster idl	9.5	17
fastest idl	6.5	10
char [ ]	6	9

Table 5: CORBA push model: Redefining the structure in monitorframe.idl clearly improves CPU usage. The faster idl consisted of removing an array of structures in favor of three base-type arrays, and the fastest idl removed an array of unions in favor of an array of doubles.

4.5. Comparison of ORBacus Notify and CORBA push method

The plan for the monitor system currently consists of an aggregate frame size of 1 MB from all 50 suppliers and a 3.2 GHz Pentium-4 processor for the ACC. All of the above numbers were therefore scaled to determine what the CPU usage would be for these specifications.

	Test	Total CPU (%) for struct	Total CPU (%) for CORBA::Any	Total CPU (%) for notserv
no CORBA	sockets w/ char[ ]	0.6	N/A	N/A
CORBA push	monitorframe.idl	2.4	5.8	N/A
	faster idl	2.0	3.5	N/A
	fastest idl	1.4	2.1	N/A
	char [ ]	1.3	1.9	N/A
Notification	monitorframe.idl	N/A	7.1	13.5
	Pipeline + Monitor	N/A	7.4	13.8

Table 6: Comparison of CORBA push model vs. ORBacus Notify.

5. Pitfalls

This section goes over some of the things that went wrong when trying to work with ORBacus Notify (and ORBacus in general) that are not related to the previously discussed issues.

- **Define hostname for the machine correctly**: This may seem obvious, and it's probably not a problem that you'll run into unless you're trying to connect to a network using DHCP. The machine's IP address must agree with the assigned hostname, whether through DNS or via the /etc/hosts file. This goes for all machines that are also trying to connect to the Notification Server (or Naming Server, or IMR). If the hostname is not set correctly, you will most likely see no connection to the IMR, or possibly a request for a RootContextPOA and nothing else.
- **Problems exist when data transfers exceed network capabilities**: What we have noticed is that when you begin to approach the tranfer capabilities of the network, latency time increases to the limit of the Timeout QoS setting, and events are lost. The exact reason for this has not been explored thoroughly.

6. Conclusion

Despite the fact that the CORBA push model outperforms the ORBacus Notify, it was determined that we would nevertheless use the latter. The primary motivations for this is that much of our existing code is based

on Notification and switching to a CORBA push model would involve a significant coding effort. Also, the ability to filter events for different consumers as well as the ability to add multiple consumers and multiple channels make it flexible enough for future extension.

The CPU budget proposal for the ACC allows for 10-15% CPU usage for CAM. To accomplish this, it will be necessary to move the Notification server off of the ACC to another machine. Our benchmarks were conducted in this fashion and showed that such a configuration could deal with the current suggested aggregate data size (1 MB) on a 100 Base-T connection.

REFERENCES

Steve Scott and N. S. Amarnath, *CARMA Monitoring System (CAM) Design*, CARMA WP Document (2003)

Iona Technologies Inc, *ORBacus Notify*, (2001)

Object Management Group, *Notification Service Specification*, (Aug 2002)