HOW DIFFERENT MESSAGING SEMANTICS CAN AFFECT SCA APPLICATIONS PERFORMANCES

Steve Bernier (Communications Research Centre Canada, Ottawa, Ontario, Canada; steve.bernier@crc.gc.ca); Hugues Latour (Communications Research Centre Canada, Ottawa, Ontario, Canada; hugues.latour@crc.gc.ca); Juan Pablo Zamora Zapata (Communications Research Centre Canada, Ottawa, Ontario, Canada; juan.zamora@crc.gc.ca)

ABSTRACT

Software Communications Architecture (SCA) compliant radios typically contain a large number of software components. Some software components provide access to hardware devices while others perform signal processing. By interacting with each other, the software components implement a radio communications standard. To interact, the software components use a middleware called Common Object Request Broker Architecture (CORBA).

Using CORBA, each interaction is carried out as an exchange of messages between two components. CORBA supports two main type of messaging: one-way and two-way. The application programming interfaces being used for Joint Tactical Radio System (JTRS) radios rely on two-way messaging. This paper explores the differences between the two types of messaging and provides performance metrics. The paper also describes design approaches that can be used to avoid common pitfalls associated with the use of both types of messaging.

1. INTRODUCTION

SCA waveform applications are typically composed of a number of software components through which voice or digital data samples travel. Typically the software components get data samples from a device, transform the data via signal processing, and send the modified data to another device. In short, SCA waveform applications are structured as a pipeline of components processing data samples.

Each software component performs a specific transformation on the data samples it receives via an input port and sends the modified data to another component via an output port. The more software components an application has, the more connections between components will be required which will lead to more interactions via the middleware. CORBA offers two main types of interactions. This paper describes both messaging types in section 2. Section 3 describes the very common empty pipeline problem which is related to the use of two-way messaging. Section 4 describes how one-way messaging can address the empty problem issue. It also describes the drawback of one-way messaging with respect to order of interactions. Section 5 presents two solutions that can preserve the order of interaction which is important for waveform applications. Finally, section 6 provides performance metrics for different type of messaging. The conclusion of the paper is provided in section 7.

2. CORBA MESSAGING

Using CORBA, the invocation of a member function implemented by an object is carried out as a message sent from a client object to a server object. When the invocation of the member function produces a result, a second message is used to communicate the result back from the server to the client object. This type of interaction is called two-way messaging. With two-way messaging, the thread of the client used to make the invocation is blocked until the return message is delivered. This means the client's execution thread is suspended while the message travels through the transport to the server, remains suspended while the server member function is invoked and the result is returned to the client via the transport.

The second type of messaging supported by CORBA is called one-way messaging. It is used when the client does not require a result back from the server. With one-way messaging, the execution thread of the client resumes before the member function is invoked on the server side. One-way messaging is often mistakenly thought to be the same as an invocation to a C/C++ function defined as having a void return value. When a client invokes a C/C++ void function, its execution is suspended until the function is executed and returns. And such a behavior actually corresponds to a two-way invocation.

The CORBA specification [1, 2, 3] actually defines four types of one-way messaging. They differ in the level of synchronization for interactions between clients and servers. The desired level of synchronization can be selected by

changing a property of the Object Request Broker (ORB) called SyncScopePolicy to one of the following values:

- **SYNC_NONE**: The client's invocation thread only blocks until the request message is created and pushed to the ORB. The invocation thread resumes before the ORB sends the request message to the transport protocol (see Figure 1). The client has no guarantee the ORB has been successful in transferring the request to the transport protocol stack on the client side.

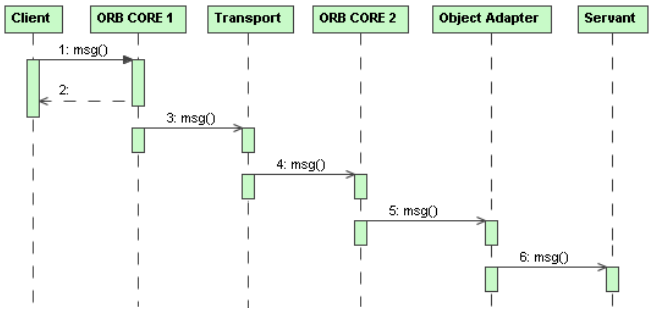

Figure 1. SYNC_NONE CORBA Request.

- **SYNC_WITH_TRANSPORT**: The client's invocation thread blocks until the ORB request message is accepted by the transport protocol stack (see Figure 2). The invocation thread is unblocked without any guarantee the request message has been received by the server.

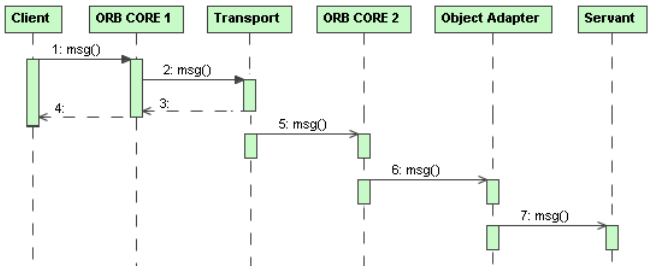

Figure 2. SYNC_WITH_TRANSPORT CORBA Request.

- **SYNC_WITH_SERVER**: The client's invocation thread blocks until the request is accepted and validated by the ORB on the server side (see Figure 3). The server-side ORB makes sure the request is for a valid function of an existing object. If the request is invalid, the acknowledgment message sent by the server will cause an exception to be raised on the client-side which will unblock the invocation thread.

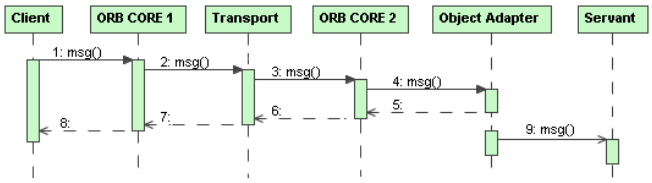

Figure 3.SYNC_WITH_SERVER CORBA Request.

SYNC_WITH_TARGET: The client's invocation thread blocks until the function is executed on the server side (see Figure 4). The sever-side ORB returns an acknowledgment message which contains no data if everything went well. The return message contains an exception otherwise. This level of synchronization provides a messaging semantic that is equivalent to two-way messaging. The difference is that two-way messaging can return a user-defined response or exception while one-way messaging cannot.

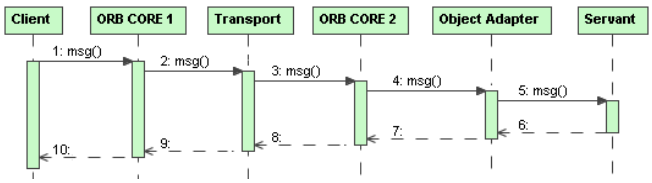

Figure 4.SYNC_WITH_TARGET CORBA Request.

Note that the default for one-way messaging is set to SYNC_WITH_TARGET for many ORBs. In fact, some ORBs don't implement SYNC_NONE and define it to be the same as SYNC_WITH_TARGET [2]. Also note that the characteristics of a transport protocol can influence the synchronization scope. For instance, The Integrity® operating system offers an Inter-Process Communication (IPC) messaging framework called "integrity connections" [4]. This IPC works in a way that the client sending a message is blocked until the server accepts the message. With such a transport, SYNC_WITH_TRANSPORT behaves the same way SYNC_WITH_SERVER does. Another example is with the use of a UDP-like transport. With such a transport, there might not be a significant difference between SYNC_NONE and SYNC_WITH_TRANSPORT since messages can be lost over the network.

3. THE EMPTY PIPELINE PROBLEM

Most SCA Applications [5, 6] are made of several software components. And often those components are interconnected in a sequence much like a pipeline where the output of the first SCA component is fed to the input of the next component and so on. Figure 5 illustrates a pipeline with four components named R1, R2, R3, and R4. The components represent the stages of the pipeline.

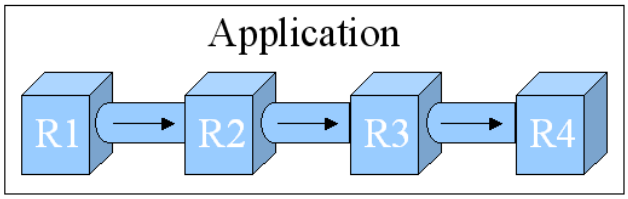

Figure 5. Processing Pipeline.

Figure 6 provides a sequence diagram of two-way interactions between the four same components processing two packets of data samples. Message number 1 shows that

component R2 receives the first data packet from R1. Message number 2 indicates that R2 performs a transformation of the input data and produces output data which is then sent to R3. Component R3 does the same and the modified data eventually reaches R4. The same sequence of interaction happens again starting at message number 10 for data packet number 2.

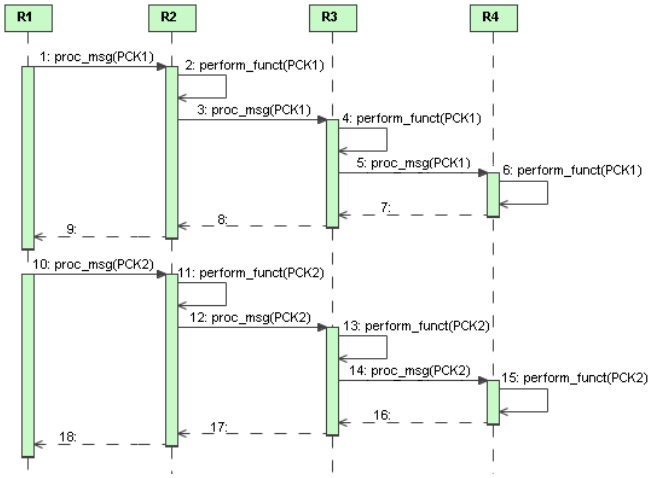

Figure 6. Two-way Messaging.

This sequence diagram clearly illustrates the pipeline of components is only working on 1 packet at a time. That's because R1 waits for all other components to be done before it can push a new data packet in the pipeline. Because of the two-way semantic is used, R2 sits idle waiting for R3 to return control. And the same is true between R3 and R4. At any one time, only one component is processing data samples. This type of interaction between the components is called the empty pipeline problem and leads to a very inefficient use of the computational elements of a platform (GPP, DSP, FPGA). This problem is the same as with multi-stage pipeline micro-processors; every functional unit in the pipeline (e.g. stage) must stay busy to maximize the usage of the processor.

4. USING ONE-WAY MESSAGING TO AVOID THE EMPTY PIPE LINE PROBLEM

To avoid the empty pipeline problem, each individual component must be able to work in parallel. In the context of an SCA Application, the solution is to make each component work on a different data packet at the same time. This can be achieved using a few different approaches. One approach consists in using one-way messaging. Figure 7 illustrates a sequence diagram where the components are using one-way messaging.

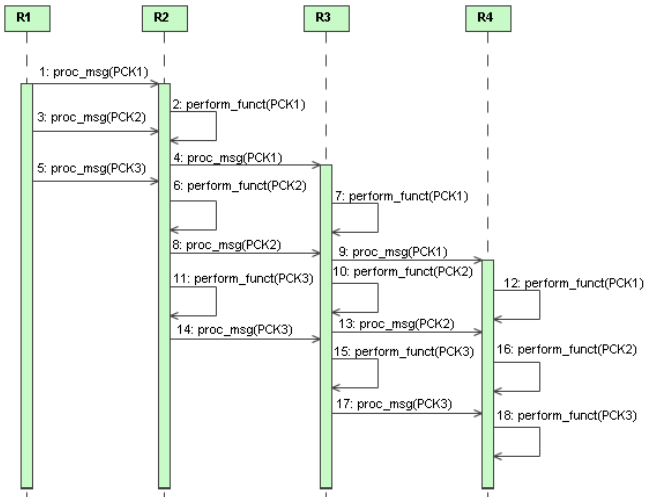

Figure 7. One-way Messaging.

This sequence diagram shows that using one-way messaging; the different components are working in parallel on three different data packets. While R4 is working on packet number 1 (message 12), R3 is working on packet number 2 (message 10), and R2 is working on packet number 3 (message 11). This approach leads to a better usage of the computational elements provided by a platform. However, the pipeline will be fully occupied only if each component takes about the same time to perform its signal processing. If one component takes more time to process its input data, it becomes a bottleneck and the remaining components in the pipeline will spend more time waiting for input data. This illustrates how crucial it is to perform a good functional decomposition of a waveform into individual components.

4.1 THE IMPACT OF ADDRESS SPACE COLLOCATION ON ONE-WAY MESSAGING

It is important to note that one-way messaging can behave like two-messaging under special circumstances. And in such a case, it will lead to the empty pipeline problem.

For a client to invoke a function implemented by a CORBA server, it must use the local stub that represents the remote server. The client invokes the function on a local stub which generates a request message and tells the ORB core to transmit the message to the targeted server using the appropriate transport layer. The stub is generated from the IDL definition of the remote function being invoked.

However, real-time ORBs use several optimizations to accelerate interactions. One common optimization allows a client and a server to transparently interact with each other directly when they are located in the same address space. This means the interaction will not cause a request to be sent over a transport, but it will result in a direct call to the

function implemented by the server. The performance improvement with such an approach is significant [7]. In the context of the SCA, address space collocation can be achieved via the use of a ResourceFactory.

The direct function call optimization is implemented in the stub. The stub is in a position to recognize that the remote object it represents is in the same address space as the client. When both the client and the server are in the same address space, the stub can perform a direction function call. This is thought to be transparent to the client since the client always uses a stub to make invocations.

However, when the stub makes a direct method invocation instead of asking the ORB to go through the transport, it does so using the client's thread. The execution of the client's thread making the invocation ends up waiting until the function returns which is the semantic of two-way messaging. Since, the ORB core is by-passed with direct function calls, the resulting messaging semantic will always be two-way even if the IDL definition specifies one-way.

However, in most cases, the use of a single address space and direct function calls will provide better performance than the use of multiple address spaces with function calls that go over a transport. That is true even if the single address space calls are done in a way that produces the empty pipeline problem. And as discussed in section 5.2, there is a solution to avoid the empty pipeline problem even with two-way messaging and it applies to the use of a single address space.

4.2 THE PACKET REORDERING PROBLEM

Using one-way messaging in a pipeline configuration can however lead to packet reordering. In other words, data packets being sent by the first component in the pipeline might be reordered before they reach the last component of the pipeline. This is not the case when the pipeline is implemented using two-way calls. With two-way calls, if component R1 sends packet number 1 before packet number 2, component R2 will always receive the packets in that same order. This is very important to most waveform applications since they use signal processing algorithms that are sequential in nature. That is, the algorithms transform data samples using information gathered from previous data samples. The reordering of CORBA interactions can happen for a number of reasons.

Transport reordering: The transport used between components can have an influence over packet reordering. Using a UDP-like transport can cause data packets to travel via different paths between a client and a server. But in the context of the SCA, embedded platforms are used. And transports on such platforms are usually more reliable.

Client-side reordering: packet reordering is mostly due to multi-threading. It can be caused when a client uses multiple threads to invoke the same server-side function. The reason is that there is no guarantee the client threads will run in the order they have been started. Furthermore, using a multi-core processor can actually cause reordering even with a fair scheduler and signal processing algorithms that use a constant amount of time to process data. That is because, in a multi-core processor, each thread can run on a different core and be affected by how busy each core is. Thus, once more, there is no guarantee client threads will run in a specific order.

The ORB can also cause packets order to be changed if it uses a thread to decouple the invocation to a stub from the introduction of the function call request on the transport. This approach can be used by ORB-generated stubs to implement the SYNC_NONE policy. However it allows a client to quickly invoke the same function which will cause several threads to be created in the client-side ORB. And as explained earlier, there is no guarantee the operating system scheduler will preserve the order of execution of the threads.

Server-side reordering: The most common problem of packet reordering is caused by servers that use multiple threads to execute the function that is invoked to transform data packets. As described above, the operating system scheduler can skew the packets order by allowing some threads to get more time slices than others.

In fact, even with a fair scheduler from a good real-time operating system, the use of multiple threads can lead to packet reordering. For instance, packets order can be changed when the amount of time necessary to perform the signal processing is not constant. In some cases the time required to perform the signal processing is related to the input data. Some algorithms use dictionaries to compress or encode data and the amount of time used to perform their task depends on the correlation between the input data and the dictionary. This means that in a multithread environment, the threads used to encode the data that finds the most matches in the dictionary will finish before the other threads, and thus the order of packets cannot be preserved.

Finally, it is important to note that most ORBs use multiple threads. CORBA objects are serviced by multiple threads unless explicitly configured otherwise. Different ORBs use different multithreading algorithms to read a message request and perform the requested invocation. Most ORBs will allow several threads to run in parallel after they have read a request from the transport. This allows multiple threads to be performing the same invocation at the same time which introduces the risk for threads to be reordered which translates to packet reordering.

With CORBA, it is possible to specify that an object must be served by only one thread at a time. This can be achieved using a specific threading policy [1, 2, 8]. This policy means the server-side ORB cannot invoke the functions of an object using multiple threads at the same time. It provides multi-thread safety for the target object. One might think that using the single thread policy would preserve the packet order by not allowing more than one thread at once to run the data processing function. But it is not the case. The ORB can still use several threads to read from the transport. The single thread policy only guarantees that the ORB threads trying to invoke the requested function will run one after the other. And the operating system scheduler can still reorder the waiting ORB threads and cause packet reordering.

5. AVOIDING DATA PACKET REORDERING

In the end, there are solutions to preserve the order of data packets. First, a client needs to make sure it does not reorder packets right from the beginning. The easiest way to do this is to use a single thread to make invocations to a same server-side function. This solution is independent of the type of messaging being used. However, on the server-side, the solution can be more or less difficult to implement. It depends on the type of messaging being used.

5.1 ONE-WAY MESSAGING

Since one-way messaging can always cause packet reordering, the solution involves stamping each packet with a sequential number as they are produced and introduced into the pipeline of components. This must be implemented at the application-level. With this solution, each component must store the packet(s) that are out of sequence in a buffer and process them in order. The components also have to deal with the possibility of having to skip packets when they don't arrive within a specific amount of time or risk delaying the processing for too long. Determining the appropriate buffer sizes and time delays is not easy and is platform-specific.

Flow control is also very important with one-way messaging. The producer of data packets can outpace the pipeline of processing components. And since buffers cannot be of unlimited capacity, flow control is required. Flow control must provide APIs that will deal with both buffer overflow and buffer underflow. The APIs must allow a server component to tell a client component to stop sending data packets when the server-side buffer is near full. The APIs also need to allow the server to tell the client to resume sending packets when the buffer is near empty. Calibrating flow control can be difficult. It involves finding the appropriate low and high buffer thresholds for each component of the pipeline. This thresholds can change with different operating environments.

Note that flow control alone generally cannot be used to avoid packet reordering. This is related to the fact that even when the flow of packets is under control, it is possible for a server-side ORB to use multiple threads and cause reordering. The only way to avoid packet reordering with flow control is to only allow a packet to be delivered to a component after it is done processing the previous packet. This kind of flow control would increase the amount of signaling required for each packet which would most certainly have a negative impact on performance.

5.2 TWO-WAY MESSAGING WITH USER-THREAD

Another approach to avoid packet reordering is to use two-way messaging. But as discussed earlier, this solution can cause a pipeline to remain empty and lead to performance issues. There is however an obvious solution to empty pipeline problem. The solution consists in using a thread within each component to decouple the reception of a packet from the processing of a packet. That is, instead of processing the data packet on the ORB thread making the invocation, the processing can be done on a user-thread that will also forward the packet to the next component. This approach allows the pipeline to fill in with more than one data packet at the same time. As shown in Figure 8, this approach leads to a pipeline usage that is very similar to one-way messaging when it is synchronized with the server as shown in Figure 3. The main difference is that two-way messaging implicitly preserves the order of packets.

Under this approach, packet ordering is preserved independently of the characteristics of the transport being used. Two-way messaging makes a client wait for the invocation to terminate before it can make a new invocation. As a result, the client is not able to send a new packet to the server before the server actually stores the current packet in a buffer. There is only one packet in transit between the moment a client invokes a function and the moment at which the execution of the server function is terminated. It is thus not possible to cause the reordering of packets. And this approach is independent of the transport characteristics and of the ORB's multithreading strategy.

Naturally, since a buffer is used to store packets within each component of the pipeline, flow control is still required. Consequently, the server needs to be able to tell the client when to stop sending new packets to avoid buffer overflow. However, with two-way messaging, the client and the server are synchronized. Therefore, if the client only regains control when the server has room to accept a new packet, the client cannot cause buffer overflow. And since the server never tells the client to stop producing packets, there is no need for an API to control buffer underflow. This solution still requires that appropriate buffer sizes be determined. However, it does not require the use of explicit APIs for high and low watermarks.

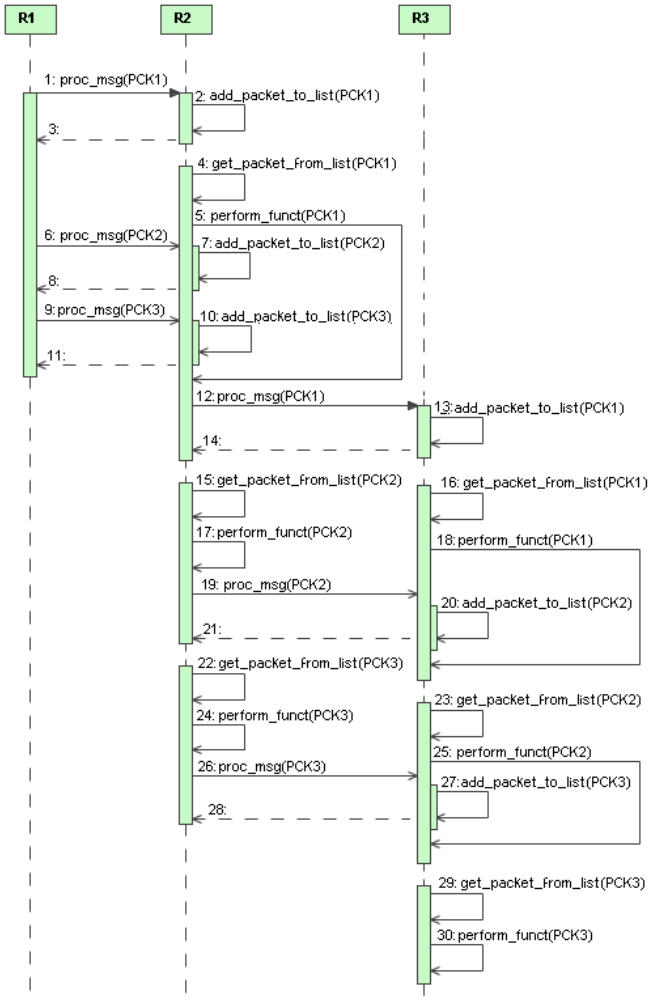

Figure 8. Two-way messaging with a user-thread

The other main difference with one-way messaging lies in the bundling of components into a single address space. As explained earlier, if components using one-way messaging are bundle together, the semantic of messaging changes to two-way messaging and leads to the empty pipeline problem. That is not the case with components that use two-way messaging with a user-thread in each component. When combined together, those components will still have a user-thread per component to perform the packet processing. And that allows different packets to be processed in parallel by different component user-threads. In other words, it allows the pipeline to be filled with packets.

6. METRICS

This section provides metrics used to make a performance comparison between the different types of messaging semantics. All the tests have been executed pushing 1000 packets through a pipeline of 3 SCA components. Each packet was made of 1024 elements of type double, which requires 8 bytes per element on the Intel Q9300 quad processor used for testing. The processor was clocked at 2.5GHz, had 6 gigs of RAM, and ran Linux FC12. The ORB used was ORBexpress RT version 2.8.2 with the IIOP transport. In every test, all the components were hosted on the same processor in different address spaces. All 3 SCA components of the pipeline perform a certain amount of signal processing that takes 5 ms. The pipeline is fed by a fourth SCA component which is the packet producer.

In theory, it should take at least 5010 ms for the 1000th packet to go through the last stage of the pipeline. It should take 5 ms for each packet to go through the first stage and the last packet should take and extra 10 ms to go through the last 2 stages. The assumption behind this reasoning is that each stage has only one thread. In other words, each stage only deals with one packet at a time.

Of course, no increase in the throughput can be achieved by simply using multiple threads in a single core processor because the threads run in time sharing mode. However, according to [9], multi-threading can be used to minimize the waste of processing cycles in a single core when many requests are made to access external memory.

With a pipeline, each stage must be able to perform in less time than it takes for the next packet to arrive. Buffers can be used to accommodate the potential bursty-ness of the traffic. Using multiple cores, different stages of a pipeline can run concurrently. If each stage is assigned to a different core, the packet budget is effectively multiplied by the number of cores [9].

Table 1 shows metrics produced with a test where the packet producer was using a one-way API to send packets to the pipeline without waiting before sending each packet. During this test, the one-way messaging semantic was set to SYNC_WITH_TRANSPORT. The table shows how much time it took for the last packet (i.e. the 1000th packet) to reach each stage of the pipeline.

	Stage 1	Stage 2	Stage 3
Time of last Pkt arrival	4463.20ms	4508.41ms	4513.61ms
# of Pkt reordered	315	520	612

Table 1: One-way messaging with a no wait producer

Interestingly enough, all 1000 packets have been processed in less than the theoretical 5010 ms for a multi-core processor hosting each stage on a different core. In fact, the last packet left the first stage after only 4463.20 ms which is before the expected 5000 ms. That is because the test allows packets to be sent faster than they can be processed and as a result, the ORB within the first stage component uses more than one thread to invoke the processing function. What happened is the core that hosted the packet producer was

periodically idle and the 3 stages where able to share the 4th core to run some extra threads concurrently. On a single core processor, this could not happen.

This first test has also demonstrated that one-way messaging coupled with the use of multiple threads at the server-side ORB has caused a significant amount of packets to be reordered. Table 1, provides the average number of packets that were received out-of-sequence in each stage of the pipeline. At the last stage of the pipeline, out of 1000 packets, over 612 packets had been reordered.

What Table 1 does not show is that the packet producer (i.e. the first component) was actually being paced by the transport. Even if the producer did not sleep before sending each packet, it periodically was blocked by the transport. Every time the transport buffers became full, the transport blocked the producer to prevent overflow. To be more precise, the stub used by the producer waited after the ORB to push the packets to the transport which was periodically blocking. The producer was blocked until the transport buffers had enough space to accept a new message. During our tests, the default buffer size for the TCP/IP stack was difficult to determine because the stack used buffer auto-tuning. This means the buffers increased in size as needed. Nevertheless, the test caused the TCP/IP stack to reach a maximum buffer size. Figure 9 shows how much time it took for the producer to send a packet to the first stage of the pipeline. When the transport buffers were not full, the producer was able to send packets in as little as 9 usec. In fact, most packets were sent in less than 16 usec. But the transport buffers reached full capacity quite often because the producer was very aggressive. Therefore, the producer periodically had to wait for approximately 44 ms.

Table 2 shows the same metrics but for a test that used one-way messaging with a packet producer that waited 5 ms between each packet being sent to the pipeline. The first thing to notice is that it took more than 5000 ms for the last packet to exit of stage 1. That is because the producer was not introducing packets faster than the pipeline stages could handle. As a result, there were fewer threads created by the server-side ORB of the stage 1 component. The different cores where not solicited enough and did not run different threads of a same stage in parallel. But as stated earlier, to some waveform applications, even a small quantity of reordering can cause serious problems.

	Stage 1	Stage 2	Stage 3
Time of last Pkt arrival	5416.57ms	5421.74ms	5426.90ms
# of Pkt reordered	95	216	349

Table 2. One-way messaging with a 5ms wait producer

Table 3 shows metrics for a test that used two-way messaging with a packet producer that did not wait between each packet it sent. With this test, the function invoked on each stage did the signal processing for 5 ms and then invoked the processing function of the next component in the pipeline. This caused the empty-pipeline problem as described in section 3 and illustrated in Figure 6.

The first thing to note about this test is that each packet went through the pipeline alone. The packets had to wait at least 5 ms at each of the three stages which adds up to 15,000 ms for 1000 packets. And since the test recorded the time at which each packet left the stages and that each stage waited after the next stage to complete, the timings are in reverse order compared to the previous tables. It took more time to finish stage 1 than it took to finish stage 2. And the same is true regarding between stages 2 and 3.

It is important to highlight that this two-way test did not cause any packet reordering. But it was done at the cost of a much lower throughput. Figure 10 shows how much time it took for the producer to send packets through the pipeline stages. The figure also shows the producer did not get blocked by the transport as often as for the one-way test (Figure 9). For the two-way test, in most cases, the producer waited for an average of 15.68 ms between packets with a couple cases were it waited more than 40 ms with one extreme case at around 85 ms.

	Stage 1	Stage 2	Stage 3
Time of last Pkt arrival	15,684.19ms	15,684.06ms	15,683.93ms
# of Pkt reordered	0	0	0

Table 3. Two-way messaging with a no wait producer

Table 4 shows the result of a test that used two-way messaging with a user-thread in each stage to decouple the reception of a packet from the processing and forwarding of the packet to the next stage. That test was executed with a producer that did not wait between the sending of each packet. Upon the reception of a new packet, instead of processing the packet before returning control to the client, each stage stores the packet in a buffer, notifies a user-thread, and returns. The user-thread wakes up when notified and takes the oldest packet from the buffer, transforms it, and sends it to the next stage.

Since the component sending the packet is blocked on the two-way call until the packet is accepted and stored in a buffer, there is never two packets in transit at the same time between two stages. This means none of the components have more than one server-side ORB thread waiting to invoke the processing function. This prevents packets from being reordered. For as long as the user-thread processes the packets in order, no reordering is caused.

This approach effectively preserves the order of packets and keeps the pipeline busy with different packets. It provides much better performance than the simple two-way approach with suffers from the empty-pipeline problem. In fact, the performances are slightly better than those of the one-way messaging approach with a time-based paced producer (Table 2). That is because using two-way messaging with a user-thread, each stage can introduce a new packet as soon as the next stage unblocks. The time between packets can be smaller than 5 ms when the stages benefit from multi-core processing.

Figure 11 shows that the use of a user thread with two-way messaging led to an average wait per packet of around 5.38 ms. It also produced fewer long waits than for the two other methods. Only a few packets took more than 20 ms to be transmitted with the extreme cases of around 60 ms.

	Stage 1	Stage 2	Stage 3
Time of last Pkt arrival	5286.22ms	5267.73ms	5297.16ms
# of Pkt reordered	0	0	0

Table 4. Two-way messaging with a no wait producer and a user-thread per stage

7. CONCLUSION

CORBA offers two types of messaging semantics: one-way and two-way. The main difference between the two resides in the level of synchronization between a client and a server which has an impact on the speed of interactions. One-way messaging is often considered a better approach than two-way messaging from a throughput perspective. However, one-way messaging cannot preserve the order of interactions between components which translates into the reordering of data packets as they flow through a pipeline of components. The ordering of packets is very important for the type of signal processing performed by waveform applications. On the other hand, two-way messaging preserves the order of interactions but leads less than optimum throughput.

Different approaches can be used to preserve the order of packets. If one-way messaging is used, the components of an application must implement extra functionality to preserve the packet order and to perform flow control. Each component must be prepared to receive packets out of order and to reorder them before performing the signal processing. Each component must also provide buffers to hold packets and implement mechanism to pace the producer to avoid buffer underflow as well as buffer overflow. This requires the use of low and high watermarks that must be tuned for every different platform the waveform is be ported to. But this solution provides much better performance than the plain two-way messaging.

Another approach is to use two-way messaging with a user-thread to decouple the reception of a packet from its processing and forwarding. The advantage of this approach is that it preserves the order of packets without stamping each packet with a number and without having to put the packets back in order after their reception. And according to preliminary tests performed, this approach is very similar with the theoretical best performance possible in a multi-core environment with as many cores as there are stages in the application. It's also very close in performance to plain one-way messaging which has the disadvantage of causing a large amount of packet reordering. Also, using this approach during testing, the metrics show that components were well synchronized with each other regarding the throughput of the data flow.

In conclusion, there are a number of things that can be done to improve the throughput of interactions between SCA components. Using a ResourceFactory to provide a single address space to host several components can provide tremendous improvements. The type of messaging used can also provide serious improvements. And, as discussed in this paper, one-way messaging is not necessarily a superior solution.

8. REFERENCES

[1] Object Management Group, *Common Object Request Broker Architecture: Core Specification*, version 3.0.3, formal/04-03-01, March 2004.

[2] Objective Interface Systems Inc., *CORBA Programming using ORBexpress RT for C++*, version 2.6, April 2006.

[3] S. Vinoski, "New Features for CORBA 3.0", Communications of the ACM, October 1998.

[4] *INTEGRITY Kernel Reference Guide*, June 2006.

[5] *Software Communications Architecture Specification, Version 2.2.2.*, December 2006.

[6] F. Lévesque, S. Bernier, "Interconnection SCA Applications", SDR'07, Denver, USA, November 2007.

[7] S. Bernier, C. Auger, J.P. Zamora Zapata, H. Latour, M. Michaud-Rancourt, "SCA Advanced Features – Optimizing Boot Time, Memory Usage, and Middleware Communications", SDRF'09 Technical Conference, 2009.

[8] M. Henning, S. Vinoski, "Advanced CORBA Programming with C++", Addison Wesley, February 1999.

[9] Y. Zhang, K. Ootsu, T. Yokota, T. Baba, "Clustered Communication for Efficient Pipelined Multithreading on Commodity MCPs", IAENG International Journal of Computer Science, 2009.

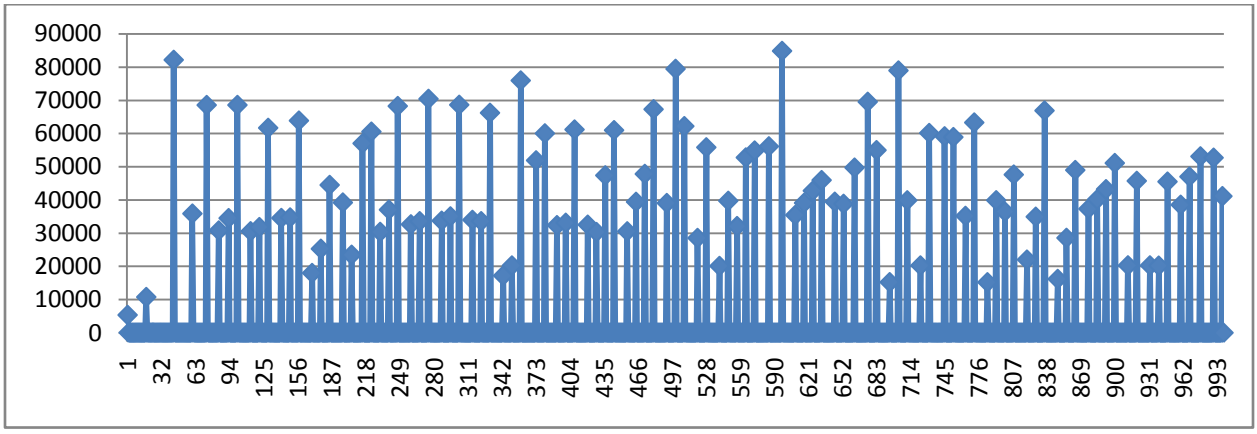

Figure 9. Time it took for the no-wait producer to send each packet using one-way messaging

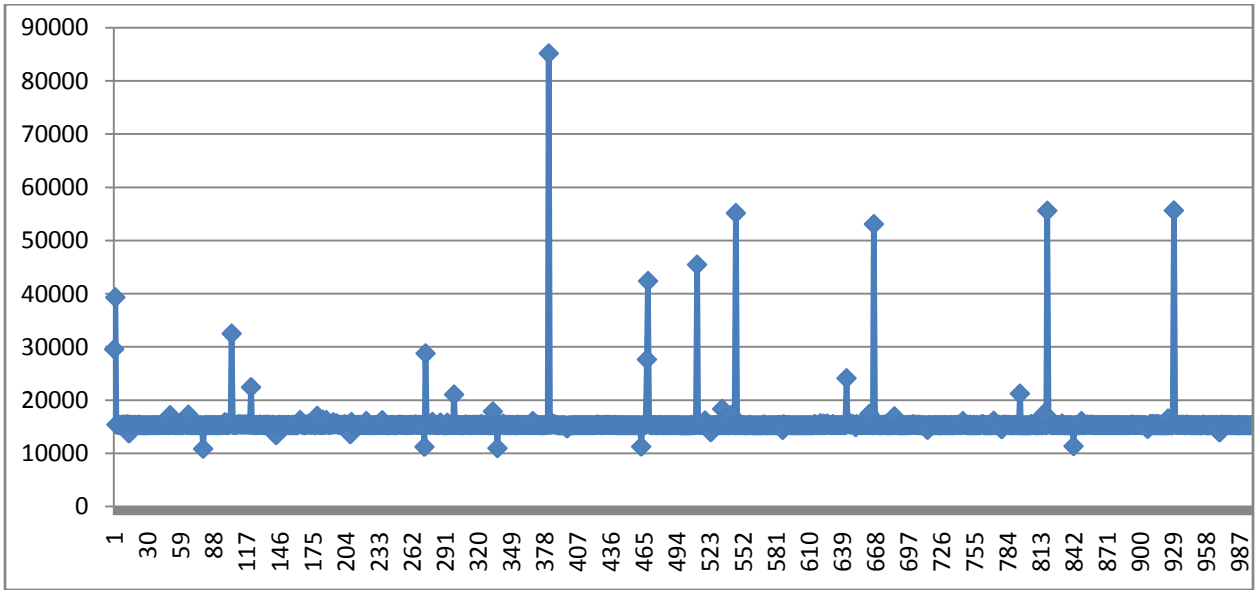

Figure 10. Time it took for the no-wait producer to send each packet using two-way messaging

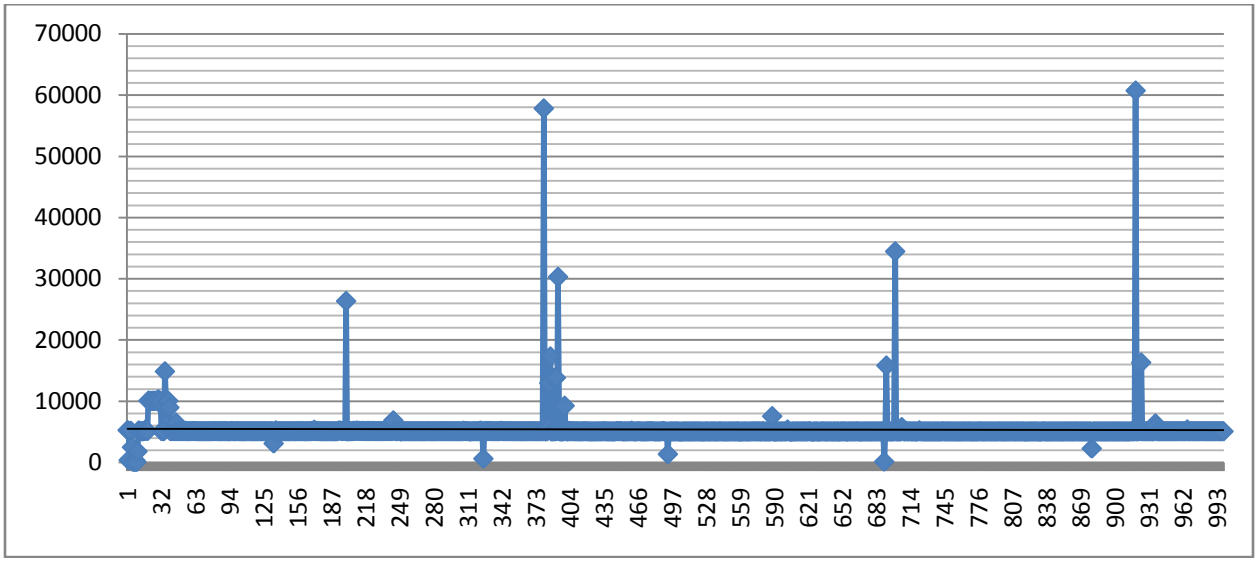

Figure 11. Time it took for the no-wait producer to send each packet using two-way messaging with a user-thread

Copyright Transfer Agreement: The following Copyright Transfer Agreement must be included on the cover sheet for the paper (either email or fax)—not on the paper itself.

"The authors represent that the work is original and they are the author or authors of the work, except for material quoted and referenced as text passages. Authors acknowledge that they are willing to transfer the copyright of the abstract and the completed paper to the SDR Forum for purposes of publication in the SDR Forum Conference Proceedings, on associated CD ROMS, on SDR Forum Web pages, and compilations and derivative works related to this conference, should the paper be accepted for the conference. Authors are permitted to reproduce their work, and to reuse material in whole or in part from their work; for derivative works, however, such authors may not grant third party requests for reprints or republishing."

Government employees whose work is not subject to copyright should so certify. For work performed under a U.S. Government contract, the U.S. Government has royalty-free permission to reproduce the author's work for official U.S. Government purposes.