HOW DIFFERENT MESSAGING SEMANTICS CAN AFFECT SCA APPLICATIONS PERFORMANCES: A BENCHMARK COMPARISON

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 types of messaging: one-way and two-way. 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 pipeline 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 types 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, and is used by the Joint Tactical Radio System (JTRS) in its application programming interfaces. With two-way messaging, the client thread 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, and 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.

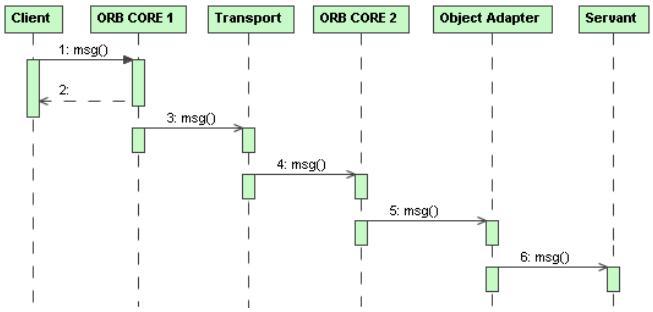

Figure 1. SYNC_NONE CORBA Request.

- **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.

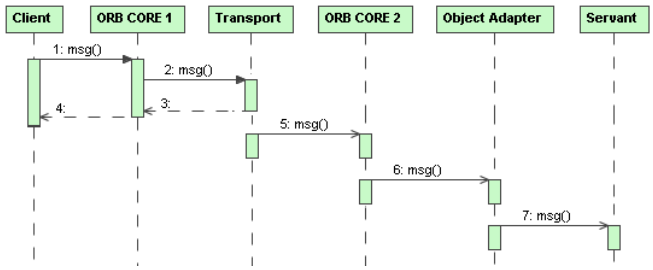

Figure 2. SYNC_WITH_TRANSPORT CORBA Request.

- **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.

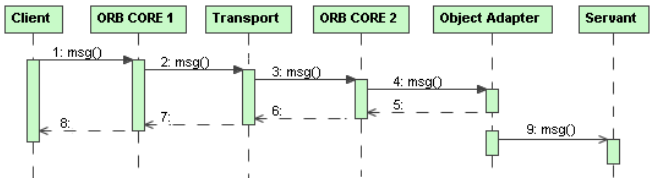

Figure 3.SYNC_WITH_SERVER CORBA Request.

- **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.

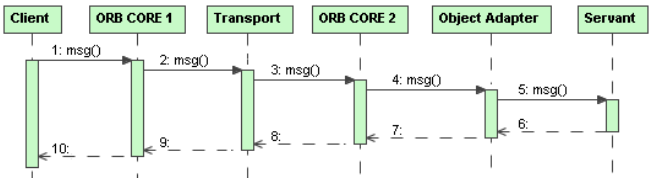

Figure 4.SYNC_WITH_TARGET CORBA Request.

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 as 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.

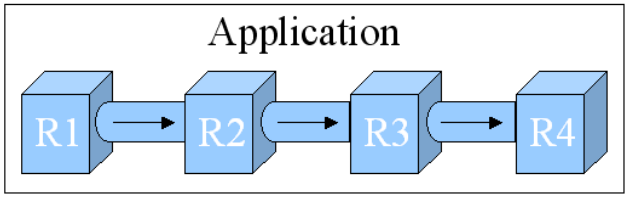

Figure 5. Processing Pipeline.

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. Messages number 2 and 3 indicate 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.

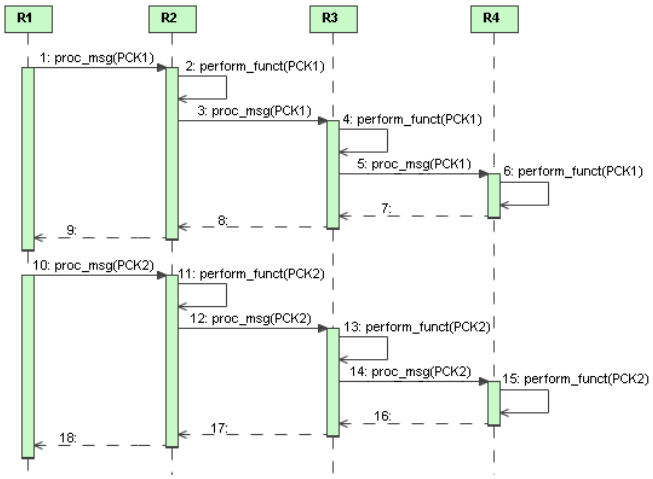

Figure 6. Two-way Messaging.

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 messaging, 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 (i.e. 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. One approach to achieve this consists in using one-way messaging. Figure 7 illustrates a sequence diagram where the components are using one-way messaging.

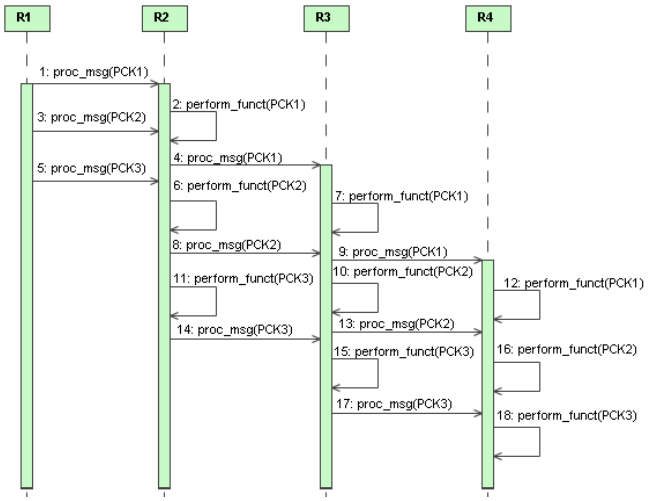

Figure 7. One-way Messaging.

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-way 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 CORBA Interface Definition Language (IDL) [7] 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 [8]. 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, and therefore 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.

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. With one-way messaging, 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. Transports on such platforms are usually more reliable, which makes this type of reordering less likely.

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; 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. 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. This time 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, 9]. 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, one 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 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.

An alternative approach considers the usage of worker-threads in one-way messaging. Unfortunately, worker threads alone cannot guarantee the absence of packet reordering as later illustrated in experiment 3 of section 6. Worker threads are further discussed in the next section as a solution to the empty pipeline problem earlier described in section 3.

5.2 TWO-WAY MESSAGING WITH WORKER-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 a solution to the 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 worker-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.

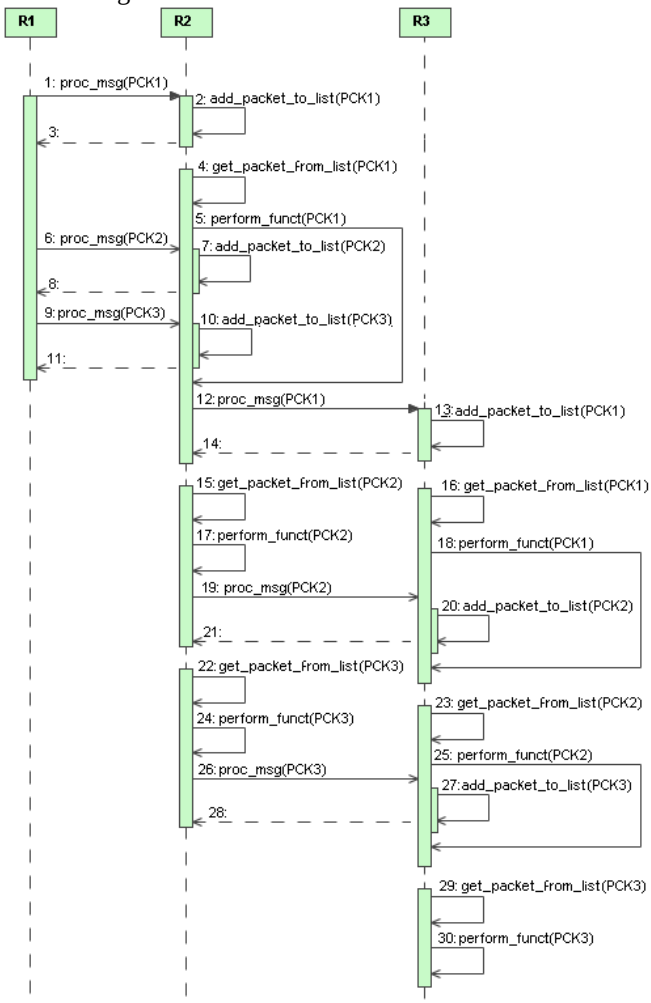

Figure 8. Two-way messaging with a worker-thread

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 co-located in an address space, the messaging semantic 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 worker-thread in each component. When combined together, those components will still have a worker-thread per component to perform the packet processing. And that allows different packets to be processed in parallel by different component worker-threads.

6. METRICS

This section provides metrics used to make a performance comparison between the different types of messaging.

Definition of the experiments

Tests were executed on a system using a Intel Q9300 quad processor, clocked at 2.5GHz, with 6 gigabytes of RAM, and ran Linux FC12. The ORB used was ORBexpress RT version 2.8.2 with the IIOP transport. The system was setup in two different configurations: the first configuration used the four cores of the processor, while the second only used one core. The second setup was used to eliminate any parallel processing among the multiple cores of the processor.

In all experiments, 3 SCA components on a pipeline performed a certain amount of signal processing that took 5 ms. The pipeline was fed by a fourth SCA component called the packet producer. Based on the theory, it can be calculated that at least 5010 ms is needed before the 1000th packet comes out of 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 an extra 10 ms to go through the last 2 stages. The assumption behind this reasoning is that each stage is supported by only one thread. In other words, each stage only deals with one packet at a time.

In a single core processor, using multiple threads cannot significantly increase the throughput since the threads run in time sharing mode. Nevertheless, according to [10], 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. And this being the case, throughput can be increased. In the experiments conducted for this paper, there was no explicit limit imposed in the number of threads used in each stage. The number of threads used was defined by processor workload, and the default settings of the operating system and the CORBA ORB

With a pipeline configuration, 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 [10].

During the tests, two different measurements were employed for packet reordering. The first, labeled as "reordered from previous", indicates the number of instances in which a packet was out of sequence when compared with the previous packet. The second, labeled "reordered from original", indicates packets that were out of

sequence considering the original sequential sending order. In an example where packets 1 through 10 arrive in the following sequence: 1, 2, 4, 5, 6, 7, 8, 9, 3, 10, under the "reordered from previous" rule, packets 4, 3 and 10 are considered reordered. That is because packet 4 arrived after packet 2, packet 3 arrived after packet 9, and packet 10 arrived after packet 3. Under the "reordered from original" rule, packets 3, 4, 5, 6, 7, 8, and 9 are considered reordered. That is because packet 3 arrived at original order 9, packet 4 arrived at original order 3, etc.

All experiments consisted in pushing 1000 packets through the pipeline of three SCA components. Each packet was made of 1024 elements of type double, requiring eight bytes per element on the Intel Q9300 quad processor used for testing.

For this paper, six different experiments were conducted that considered three variations. The first variation involved changing the CORBA messaging semantics from one-way to two-way. During this test, the one-way synchronization level was set to SYNC_WITH_TRANSPORT. The second variation consisted in making the packet producer wait 5ms or not between each packet being pushed in the pipeline. The 5ms wait time represents a typical amount of time allocated to processing a packet for waveform applications. The third variation consisted in making each stage of the pipeline use a separate worker thread to process the incoming packets or not. An example of using a worker thread can be seen in figure Figure 8, where the client's invocation thread unblocks in messages 3, 8 and 11, while a worker thread retrieves and process packets starting in message 4.

The experiments can be summarized in Table 1. Each experiment was run under both the multi-core and single core configurations.

Experiment #	Message mechanism	Producer wait time	Worker thread
1	one-way	0	no
2	one-way	5 ms	no
3	one-way	0	yes
4	one-way	5ms	yes
5	two-way	0	no
6	two-way	0	yes

Table 1: Experiment configurations

Experiment 1

Table 2 shows metrics produced for the Multi-core setup, with a test where the packets are exchanged between the pipeline components using a one-way producer, used a one-way API to send packets to the pipeline, and without waiting before sending each packet. The table shows how much time it took for the last packet (i.e. the 1000th packet) to leave each stage of the pipeline.

	Packet Producer	Stage 1	Stage 2	Stage 3
Time of last Pkt sent/processed	4330.24ms	4458.38ms	4477.53ms	4498.38ms
# of Pkt reordered				
with previous	-	252	363	487
with original	-	358	564	694

Table 2: One-way messaging with a no-wait producer (4 cores)

Results from Table 2 provide several interesting facts: First, it can be observed that the packet producer sent the last packet after only 4330.24ms, well before the expected 5s mark. That was because the test allowed for packets to be sent faster than they could be processed, and as a result, the ORB within the first stage component used more than one thread to invoke the processing function. Second, all 1000 packets were processed in less than the theoretical 5010ms. The third stage completed processing of the last packet after only 4498.38ms. The reduced time can be explained by the fact that all four cores of the processor were used for processing packets. The core that hosted the packet producer was periodically idle and the 3 stages were able to share the 4th core to run extra threads concurrently. On a single core processor, this did not happen.

Table 2 also provides the number of packets that were received out-of-sequence in each stage of the pipeline. It shows that packet reordering was substantial, and it only increased with each processing stage. At the last stage of the pipeline, out of 1000 packets, over 487 were out of sequence with respect to the previous packet number, and 694 did so when compared with the original sequential order. One-way messaging, coupled with the use of multiple threads at the server-side ORB, is responsible for the significant amount of packet reordering. Under both reordering measurements, the largest amount of reordering happened at stage 1. This is because the packet producer introduced as many packets as the transport allowed causing many threads from stage 1 to process packets concurrently. The stage 1 threads were scheduled in a way that some threads finished before others that had started earlier. This is the reason several packets were processed out of sequence.

Table 2 finally shows that the packet producer was actually paced by the transport since the last packet was sent after 4330.24ms. 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. In CORBA terminology, the client stub used by the producer component to send packets to the stage 1 component had to wait for the ORB to push the packets to the transport which was periodically blocking. The producer was blocked by the transport until the transport buffers had enough space to

accept new packets. 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 packets to the first stage of the pipeline. When the transport buffers were not full, the producer was able to send some packets in as little as 6 µsec. In fact, 80% of the packets were sent in less than 15 µsec, while 90% of the packets were sent in less than 35 µsec. The average for the lowest 90% of the cases was 9.67 µsec. But the transport buffers reached full capacity quite often (10% of the cases) because the producer was very aggressive. For the highest 10% of the cases, the average wait was 42,787 µsec including one case with a maximum of 108,710 µsec

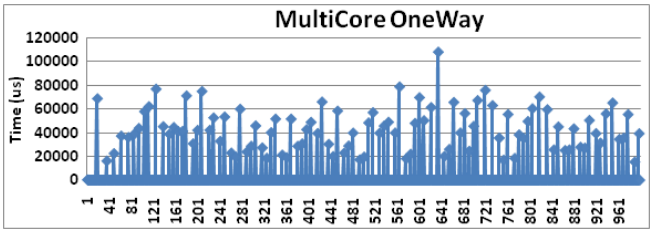

Figure 9. Time measurements for the no-wait producer to send each packet using one-way messaging (4 cores)

Table 3 presents a similar test as the one conducted for one-way messaging with a no-wait producer (Table 2), but it was obtained under the single core setup.

	Stage 1	Stage 2	Stage 3
Time of last Pkt processed	15236.05ms	15261.42ms	15280.76ms
# of Pkt reordered			
with previous	442	565	608
with original	941	936	927

Table 3: One-way messaging with a no-wait producer (1 core)

Considering that this test ran using a single core, the theoretical floor was set at 15 seconds (each stage taking 5 seconds to process its 1000 packets). Table 3 shows that the test was executed in a time above the theoretical floor. That is due to the overhead associated with moving the packets across the pipeline using the transport.

Figure 10 shows how much time it took for the producer to send packets to the first stage of the pipeline. In this scenario, the producer was also able to send some packets in as little as 6 µsec. 81% of the packets were sent in less than 12 µsec, while 89% of the packets were sent in less than 26 µsec. The average for the lowest 89% of the cases was 8.11 µsec. When the transport buffers reached full capacity (11% of the cases) the average wait was 132,475 µsec including seven cases above 200,000 µsec.

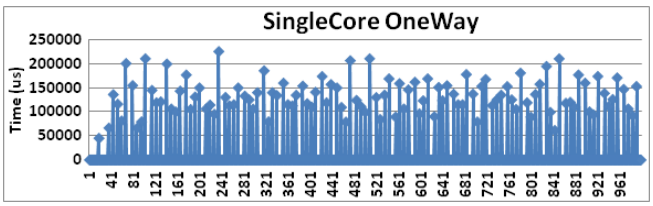

Figure 10. Time measurements No-Wait SingleCore OneWay

Packet reordering under the single core test was greater than in the multi-core test. Again, the message mechanism allowed packets to be introduced into the transport without synchronization and the multi-threading allowed individual component threads to process packets out of sequence.

Experiment 2

Table 4 and Table 5 show the same metrics but for a test that used one-way messaging with a packet producer that waited 5 ms between each packet being pushed into the pipeline. Table 4 shows the test using the multi-core setup while Table 5 does so using the single core setup.

The first thing to notice is that it took more than 5000 ms for the last packet to exit 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. Pacing the packet producer resulted in less reordering, 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 processed	5135.00ms	5194.61ms	5213.95ms
# of Pkt reordered			
with previous	174	305	471
with original	341	455	576

Table 4. One-way messaging with a 5ms wait producer (4 cores)

Table 5 shows the results of experiment 2 using a single core. When compared with results from using a no-wait producer (Table 3) there is a slight increase in processing time associated with the pacing of the packet producer. In terms of reordering, pacing the packet producer caused a slight decrease in packet reordering.

	Stage 1	Stage 2	Stage 3
Time of last Pkt processed	15099.51ms	15432.74ms	15456.94ms
# of Pkt reordered			
with previous	367	425	436
with original	753	747	747

Table 5. One-way messaging with a 5ms wait producer (1 core)

Experiment 3

Table 6 and Table 7 present the results of a test using one-way messaging with a worker-thread in each stage to decouple the reception of a packet from the processing and forwarding of the packet to the next stage. The 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 it before returning control to the ORB, each stage stored the packet in a buffer, notified a worker-thread, and returned. The worker-thread would wake up when notified and take the oldest packet from the buffer, process it, and send it to the next stage.

	Stage 1	Stage 2	Stage 3
Time of last Pkt processed	5146.50ms	5182.71ms	5720.99ms
# of Pkt reordered			
with previous	11	11	11
with original	524	524	524

Table 6. One-way messaging with a no-wait producer and a worker-thread (4 cores)

From Table 6 it can be seen that packet reordering behaved differently from previous experiments. It was observed that packets were only reordered upon arrival to stage 1, and after that, stage 2 and 3 received them in the exact order as they were processed by stage 1. The reason for such a behavior lies in the fact that the packet producer was able to use many cycles to produce and send packets. The producer produced more packets than the stage 1 component could handle. That caused several threads to be used by the stage 1 component. However, since the stage 1, 2, and 3 used the same amount of time to process packets, there was no need for multiple threads to be used in stages 2 and 3 which reduced the risk for packet reordering.

It is also worth highlighting that the introduction of a worker-thread resulted in increased total time for execution. In experiment 1 (Table 2) the last packet was processed by stage three after 4498.38ms, while the equivalent timing for experiment 3 resulted in 5720.99ms. That is due to the overhead associated with synchronization.

Figure 11 shows how much time it took for the producer to send packets to the first stage of the pipeline. In this scenario, the producer was also able to send some packets in as little as 7 µsec. 81% of the packets were sent in less than 19 µsec, while 89% of the packets were sent in less than 27 µsec. The average for the lowest 89% of the cases was 11.59 µsec. When the transport buffers reached full capacity (11% of the cases) the average wait was 1,005 µsec including one case with a maximum of 34,730 µsec.

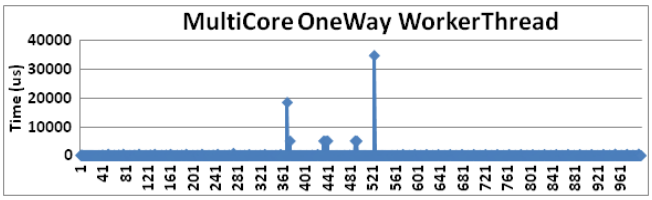

Figure 11. Time measurements No-Wait MultiCore OneWay UserThread

For the single core results illustrated in Table 7, it can be seen that even though the packet producer was not paced, there was no packet reordering. It is very important to note however that under this configuration, there is no guarantee that packets will be in sequence. The scheduler can still lead to packet reordering since the producer can introduce a new packet before the old packet is stored in the queue of the next stage component as was the case for the multi-core test presented in Table 6.

An additional interesting fact under the single core setup is that the total time per stage was only slightly increased from the one seen in Table 3. The latter indicates that the user-created thread has a minimal impact under the single core setup.

	Stage 1	Stage 2	Stage 3
Time of last Pkt processed	15328.17ms	15299.67ms	15299.61ms
# of Pkt reordered			
with previous	0	0	0
with original	0	0	0

Table 7. One-way messaging with a no-wait producer and a worker-thread (1 core)

The final item to highlight from Table 7 is how the scheduler behaved under this experiment. The Linux kernel seemed to behave in a last-in-first-out (LIFO) mode. The normal set of sequential actions each stage has to execute is as follows:

1) Receive packet
2) Process packet
3) Forward packet to next stage (stages 1 and 2)
4) Take a time-stamp since the stage is finished with all activities pertaining to the current packet

Times presented in Table 7 are those obtained at action 4. The data shows that stage 1 finished all activities for packet 1000 after stage 2 did so and stage 2 finished after stage 3. This is the result of the scheduler switching to a new task immediately after the packet is introduced into the transport (action 3) and never switching back to the time stamping (action 4) before the packet was processed by the following stages. The actual sequence observed for packet 1000 is as follows:

1) Stage 1 received message
2) Stage 1 processed message

3) Stage 1 introduced packet into the transport
4) Scheduler switched tasks giving control to Stage 2
5) Stage 2 received message
6) Stage 2 processed message
7) Stage 2 introduced packet into the transport
8) Scheduler switched tasks giving control to Stage 3
9) Stage 3 received message
10) Stage 3 processed message
11) With no one to forward the message to, stage 3 obtains current time (time stamp)
12) Scheduler switched tasks giving control to Stage 2
13) Stage 2 obtains current time (time stamp)
14) Scheduler switched tasks giving control to Stage 1
15) Stage 1 obtains current time (time stamp)

The same behavior was also observed in the single core configuration test for experiment 4 below.

Figure 12 shows how much time it took for the producer to send packets to the first stage of the pipeline. In this scenario, the producer was also able to send some packets in as little as 6 µsec. 92% of the packets were sent in less than 17 µsec. The average for the lowest 92% of the cases was 11.13 µsec. When the transport buffers reached full capacity (8% of the cases) the average wait was 1,110 µsec including one case with a maximum of 20,411 µsec, and a second case with a wait of 15,851 µsec.

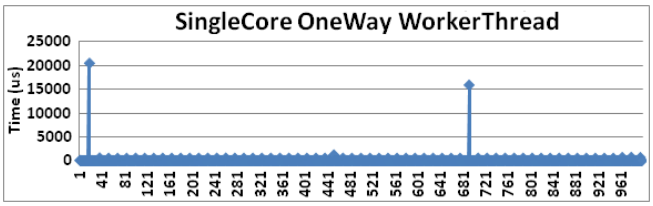

Figure 12. Time measurements No-Wait SingleCore OneWay UserThread

Experiment 4

Table 8 and Table 9 present the results of a test using one-way messaging with a worker-thread. In the experiment the packet producer waited 5ms before sending each packet.

	Stage 1	Stage 2	Stage 3
Time of last Pkt processed	5805.60ms	5810.69ms	5816.96ms
# of Pkt reordered			
with previous	0	0	0
with original	0	0	0

Table 8. One-way messaging with 5ms wait producer and a worker-thread (4 cores)

The first item to highlight from Table 8, is the fact that no reordering was observed during the experiment. The combination of explicitly pacing the packet producer combined with a user created thread at each of the stages has lowered the probability of packet reordering enough so that none of the 1000 packets were reordered. But as explained for Table 7 above, one-way messaging cannot guarantee packet ordering.

The overhead observed, calculated as the total experiment time of 5816.96ms minus the theoretical processing time of 5010ms, was of 806.96ms. This value will be compared against experiment 6 where two-way messaging is used coupled with a worker-thread.

Table 9 presents experiment 4 using a single core configuration. No packet reordering was observed, and stage times were slightly increased when compared to those in Table 5. Notice that the task scheduler behavior is similar to the one observed in the single core test (experiment 3, Table 5).

	Stage 1	Stage 2	Stage 3
Time of last Pkt processed	15511.86ms	15497.68ms	15483.59ms
# of Pkt reordered			
with previous	0	0	0
with original	0	0	0

Table 9. One-way messaging with a 5ms wait producer and a worker-thread (1 core)

Experiment 5

Table 10 and Table 11 show metrics for a test that used two-way messaging with a packet producer that did not wait between each packet being 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 notice about this test is that each packet went through the pipeline alone. In this configuration, each packet takes at least 5 ms to go through each stage which adds up to 15,000ms for 1000 packets. And since the test records the time at which each stage finished the packet processing, the timings are also in reverse order. It took more time to finish stage 1 than it took to finish stage 2. And the same is true for stages 2 and 3. Notice that this reversed order is associated with the two-way message semantics, and not with the task scheduler as seen in experiments 3 and 4 for the single core configuration.

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 when compared to the one observed for the one-way messaging from Table 2.

	Stage 1	Stage 2	Stage 3
Time of last Pkt processed	18407.82ms	18393.48ms	18379.11ms
# of Pkt reordered			
with previous	0	0	0
with original	0	0	0

Table 10. Two-way messaging with a no-wait producer (4 cores)

Figure 13 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 18000 µsec between packets. In most cases, it waited between 16000 µsec and 20000 µsec. In a few cases, it waited for 22000 µsec. In this scenario, the fastest interval in which the producer was able to send two consecutive packets was 15,260 µsec. 99% of the packets were sent in less than 21,000 µsec, with an average of 18,000 µsec. The maximum wait in this case was 22,371 µsec.

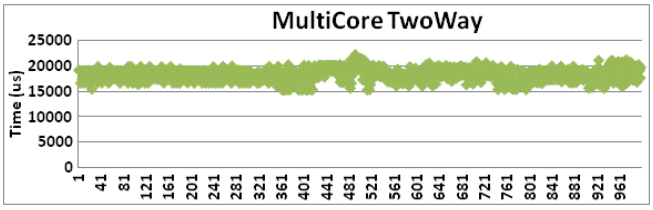

Figure 13. Time measurements No-Wait MultiCore TwoWay

Table 11 shows the results of experiment 5 under the single core setup. The overhead observed at stage 3 for this single core setup was of 273.58ms. That is calculated as 15283.58ms after stage 3 has finished all activities for packet 1000 minus the 15010 theoretical minimum for the test. It is interesting to notice how the overhead observed is substantially less than the one observed for the equivalent test under the multi-core setup (Table 10) which was of 3369.11ms. The latter is calculated as 18379.11ms after stage 3 finished all activities for packet 1000 minus the 15010 theoretical minimum for the test. The difference is caused by the underutilization of the processing cores. Tasks where repeatedly allocated to different cores and the data copied across the cores. For experiment 5 an overhead of 2 sec was eliminated by forcing the CPU affinity of all 4 components to their own core. For experiment consistency, all the data collected for the multi-core did not set CPU affinity for any components. In the case of the single core setup, as everything was performed in the same core, task switching and data movement was substantially reduced.

	Stage 1	Stage 2	Stage 3
Time of last Pkt processed	15312.17ms	15297.90ms	15283.58ms
# of Pkt reordered			
with previous	0	0	0
with original	0	0	0

Table 11. Two-way messaging with a no-wait producer (1 core)

Figure 14 shows how much time it took for the producer to send a packet to the first stage of the pipeline. In this scenario, the fastest interval in which the producer was able to send two consecutive packets was 15,220 µsec. 99% of the packets were sent in less than 15,773 µsec, with an average of 15,262 µsec. The maximum wait in this case was 19,163 µsec.

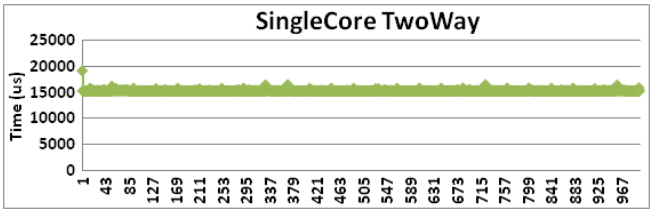

Figure 14. Time measurements No-Wait SingleCore TwoWay

Experiment 6

Table 12 and Table 13 show the result of a test that used two-way messaging with a worker-thread. In this experiment, the packet producer did not waiting between the sending of packets.

As explained earlier, since the component sending the packet is blocked on the two-way call until the packet is accepted and stored in a buffer, there are 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 worker-thread processes the packets in order and the transport does not reorder packets, no reordering is possible.

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 which suffers from the empty-pipeline problem. In fact, the timing observed in Table 12 are close to those observed in the one-way messaging approach with a time-based paced producer (Table 6). That is because using two-way messaging with a worker-thread, each stage can introduce a new packet as soon as the next stage unblocks. The time between packets can be shorter than 5 ms when the stages benefit from multi-core processing.

	Stage 1	Stage 2	Stage 3
Time of last Pkt processed	5388.72ms	5394.15ms	5399.72ms
# of Pkt reordered			
with previous	0	0	0
with original	0	0	0

Table 12. Two-way messaging with a no-wait producer and a worker-thread per stage (4 cores)

Figure 15 describes the distribution of time required by the producer to send 1000 packets, under two-way messaging with a worker-thread. This test resulted in 92% of the packets being sent in less than 100µsec, and 99% sent in less than 120 µsec. From the 1000 packets, only in three cases the producer waited for as long as 5ms and one single case, the producer waited 66ms. This scenario did not cause significant long waits as observed in Figure 9 and Figure 13.

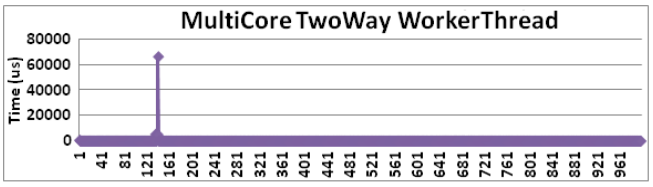

Figure 15. Time measurements No-Wait MultiCore TwoWay User Thread

Finally Table 13, presents the same execution test under the single core setup. As with the results of the multi-core setup, the two-way message semantics coupled with worker-thread produced comparable results as those produced for the one-way semantics with a worker-thread shown in Table 9.

	Stage 1	Stage 2	Stage 3
Time of last Pkt processed	15358.32ms	15372.36ms	15344.16ms
# of Pkt reordered			
with previous	0	0	0
with original	0	0	0

Table 13. Two-way messaging with a no-wait producer and a worker-thread per stage (1 core)

Figure 16 shows how much time it took for the producer to send a packet to the first stage of the pipeline. This test resulted in wait times as low as 66 µsec. 80% of the packets being sent in less than 77µsec, and 99% sent in less than 91 µsec. From the 1000 packets, only in one case the producer waited for as long as 65,198 µsec.

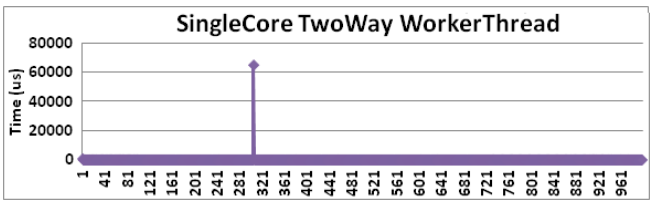

Figure 16. Time measurements No-Wait SingleCore TwoWay User Thread

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 the messaging mechanism introduces an empty pipeline problem that substantially degrades troughput.

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 a mechanism to pace the producer to avoid buffer overflow as well as buffer underflow. This requires the use of fairly sophisticated synchronization techniques (like low and high watermarks) that must be tuned for every different platform the waveform is be ported to. But this solution approach can provide better throughput than the plain two-way messaging.

Another approach is to use two-way messaging with a worker-thread to decouple the reception of a packet from its processing and forwarding. The advantage of this approach is that it can preserve the order of packets without stamping each packet with a number and without having to sort the packets back in order after their reception. Table 14 shows that this approach yielded near-optimum throughput when compared to one-way messaging using a single core, with no reorder observed for any of the communication mechanisms. From Table 14, it would be tempting to infer that the worker thread solved the reordering issue, and to conclude that by presenting a better throughput (although

only marginally), the one-way mechanism should be preferred.

		Stage 1	Stage 2	Stage 3
Time of last Pkt processed	one-way	15328.17ms	15299.67ms	15299.61ms
	two-way	15358.32ms	15372.36ms	15344.16ms
# of Pkt reordered				
with previous	one-way	0	0	0
	two-way	0	0	0
with original	one-way	0	0	0
	two-way	0	0	0

Table 14. One-way / Two way messaging with a no-wait producer and a worker-thread (1 core)

Table 15 shows a similar comparison only this time, reorder was observed for the one-way messaging. From this, it can be concluded then that the worker-thread targets and solves the empty pipeline problem, and that it cannot guarantee to solve the reordering problem. The performance of the one-way messaging was only marginally better than the two-way alternative.

In all cases the metrics show that components were well synchronized with each other regarding the throughput of the data flow.

		Stage 1	Stage 2	Stage 3
Time of last Pkt processed	one-way	5146.50ms	5182.71ms	5720.99ms
	two-way	5388.72ms	5394.15ms	5399.72ms
# of Pkt reordered				
with previous	one-way	11	11	11
	two-way	0	0	0
with original	one-way	524	524	524
	two-way	0	0	0

Table 15. One-way / Two way messaging with a no-wait producer and a worker-thread (4 core)

To conclude, among the number of things that can be done to improve the throughput of interactions between SCA components,-the type of messaging used can make a significant difference.

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] Object Management Group, *CORBA 3.0 - OMG IDL Syntax and Semantics chapter* formal/02-06-39, June 2002.

[8] 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.

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

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