A High-Performance Event Service for HPC Applications

Ian Gorton, Daniel Chavarría-Miranda, Manojkumar Krishnan, Jarek Nieplocha
Applied Computer Science, Pacific Northwest National Laboratory
{ian.gorton, daniel.chavarria, manoj, jarek.nieplocha}@pnl.gov

Abstract

Event services based on publish-subscribe architectures are well established components of distributed computing applications. Recently, an event service has been proposed as part of the Common Component Architecture (CCA) for high-performance computing applications. In this paper we describe our experiences investigating implementation options for the CCA event service that exploit interprocess communications mechanisms commonly used on HPC platforms. The aim of our work is to create an event service that supports the well-known software engineering advantages of publish-subscribe architectures, and provides performance levels approaching those achievable using more primitive message-passing mechanisms such as MPI.

1. Introduction

Event services based on publish-subscribe communications are well-established components of distributed computing applications [1]. Event services facilitate flexible inter-process communications of messages in a loosely-coupled, dynamic architecture. Distributed computing standards such as CORBA and the Java Enterprise Edition provide event services with the Event/Notification Service and Java Messaging Service respectively [2,3]. There are also many successfully deployed proprietary event services, each offering subtly different quality of service and features sets [4,5].

In 2006, an event service specification was proposed for the Common Component Architecture (CCA) [8] standard for high-performance computing applications. The event service is specified as a collection of SIDL (Scientific Interface Description Language) interfaces. This permits a wide design space of possible event service implementations that conform to the SIDL specification.

An initial implementation of the CCA event service has been built in the SciRun scientific computing environment [6]. This permits events to be exchanged between CCA components executing in the same process space. The SciRun event service is essentially part of the CCA framework (component container) that supports components executing in the same address space on SciRun.

In this paper we describe our initial efforts to design and implement a CCA-compliant event service for HPC applications. We explore the design alternatives that have been considered in order to achieve performance levels approaching those of more primitive message-passing mechanisms such as MPI. Specifically, we present the design and preliminary results from an implementation of the event service on the Cray XD1 platform exploiting ARMCI [7] communications primitives.

2. Event Services

Contemporary events services have their origins in message-oriented middleware (MOM) technologies developed in the 1990s by various vendors (e.g IBM's MQSeries; TIBCO's Rendezvous). Event service architectures are based on publish-subscribe communication mechanisms.

Publish-subscribe messaging extends the basic MOM mechanisms to support *1 to many*, *many to many*, and *many to 1* style communications. Publishers send a single copy of a message addressed to a named *topic*. Topics are a logical name for a communications channel implemented by the event service. Subscribers listen for messages that are sent to topics that interest them. The event server then distributes each message sent on a topic to every subscriber who is listening on that topic. This basic scheme is depicted in Figure 1. In terms of software engineering, publish-subscribe has some attractive properties. Senders and receivers are decoupled, each respectively unaware of which applications will receive a message, and who actually sent the message. Each topic may also have more than one publisher, and the publishers may appear and disappear dynamically. These give considerable

29th International Conference on Software Engineering Workshops(ICSEW'07)
0-7695-2830-9/07 $20.00 © 2007 IEEE

flexibility over static configuration regimes. Likewise, subscribers can dynamically subscribe and unsubscribe to a topic. Hence the subscriber set for a topic can change at any time, and this is transparent to the application code.

The event service has the responsibility for managing topics, and knowing which subscribers are listening to which topics. It also has the responsibility for delivering every message sent to all active current subscribers, and buffering messages until subscribers digest them.

In addition to basic message delivery capabilities, event services support a diverse set of additional features. These include guaranteed message delivery in the face of software/hardware failure, subscription to multiple topics using wildcarded topic names, "time-to-live" values for messages in the server, and many more that space precludes us from describing.

Due to this wide range of capabilities and close similarities in APIs, there is considerable overlap in usage scenarios between distributed event and messaging services. At one end of the spectrum, applications may use these services for communicating relatively small and infrequent event notifications. At the other end, relatively large and frequent messages may be transported. High quality service implementations strive to support all usage scenarios, for example [9] which efficiently supports message sizes from 100 bytes to 100Kbytes. Others are not designed for certain use cases and consequently may not meet performance or reliability expectations when their usage steps outside expected tolerances [4,5]

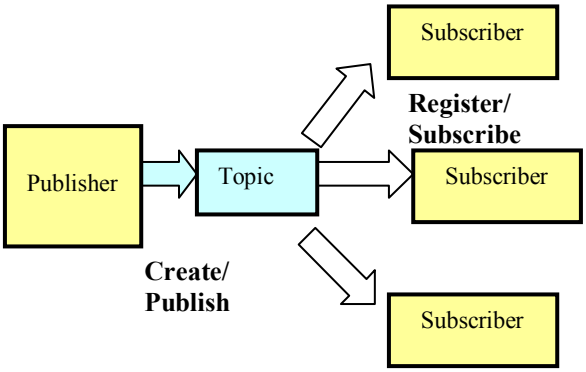

Figure 1 Event Service Architecture based on Publish-Subscribe Messaging

3. CCA Event Service

The proposed CCA event service comprises a collection of SIDL interfaces that facilitate publish-subscribe messaging using wildcarded topics. Figure 2 depicts a UML class diagram that represents the interfaces and their relationships.

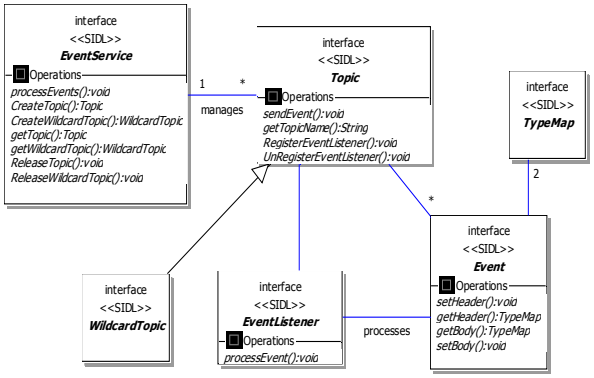

Figure 2 CCA Event Service

The key abstraction is the *EventService*, which provides methods for publishers to create *Topic* instances and subscribers to register their desire to receive messages from a named *Topic*. Subscribers must register an object that implements the *EventListener* interface with a *Topic* in order to receive messages asynchronously. Event receipt is triggered when a subscriber makes an explicit call to the *EventService*::*processEvents()* method.

The initial CCA event service implementation in the SciRun workbench assumes event publishers and subscribers exist in the same process. This simplifies event delivery and event queue management due to all components sharing a virtual address space. In an application environment where publishers and subscribers must communicate across processor and address space boundaries and achieve high performance, the event service implementation becomes much more complex.

4. HPC Event Service Design

We have created a preliminary implementation of the CCA event service running on a distributed memory HPC system. Our implementation utilizes a combination of messaging based on MPI and one-sided communication primitives based on ARMCI [7].

There were two main considerations driving our design and implementation: maintaining support for the interface and semantics of the draft CCA event service specification and achieving high performance with low overhead on distributed memory HPC clusters. One of the key design challenges we faced was maintaining the object-oriented nature of the CCA event service specification in the presence of multiple, distributed address spaces.

Our implementation is based on MPI and ARMCI, which use a process-based mechanism for executing applications on an HPC cluster. Each processor, in general, executes a single-threaded process launched from the same executable image (Single Program Multiple Data paradigm). In MPI, processes are identified by their rank from 0 to $p - 1$, where p is the total number of processes executing the application.

When executing our event service implementation on p processors, we have reserved process 0 as a special *topic directory* process. Process 0 maintains a directory of all the topics that are currently being published on the executing application and the location of the event queues that are maintained by the publishing nodes.

Publishing and subscribing processes communicate with process 0 in order to create new topics or query information for a topic via a simple MPI messaging protocol. Currently, this protocol supports four types of messages: (1) **Add** a new topic, (2) **Query** a topic for its publisher and even queue locations, (3) **Remove** a topic from the published list or (4) **Quit** servicing requests. Process 0 maintains an efficient mapping of topic names to publisher processes & event queue location (at that publisher process). We exploit ARMCI's ability of performing remote memory operations (get & put) by storing a pointer to the actual location of the topic's event queue on process 0's topic directory mapping.

```
struct TopicListEntry {
  int publisher; // publisher process ID
  EventList *eventList; // remote pointer
};

map<std::string, TopicListEntry> topicMap;
```

Figure 3: Topic Directory Entry

Figure 3 shows the declarations for the entries in process 0's topic directory mapping. Note that the `eventList` pointer corresponds to a memory address on the publisher's address space, so it can *never be dereferenced* directly on process 0.

Once the publisher process has received an acknowledgement from the topic directory process that a topic has been created, it does not need to interact any more with the topic directory process. In fact, *all* event publishing operations are local to its address space.

To minimize the need for synchronization and coordination between topic publishers and subscribers we place all published events on special ARMCI-allocated memory areas on publishing processes. These memory areas are directly accessible to other processes through ARMCI get & put calls. Subscribers to a topic can thus directly access the publishing process' memory to consume events. By allocating published events on the publishing process' memory space, we avoid the need to synchronize the state and existence of buffer space on the receiving subscriber processes as would need to be done in a traditional message-based publish-subscribe implementation.

A publisher process uses local C++ operations and methods to manipulate events on its local address space. However, there are some restrictions on how objects can be laid out in memory in order to be accessible to remote subscriber processes using ARMCI get operations. ARMCI deals with blocks of memory without any type semantics attached to them (blocks of bytes). For this reason, after an ARMCI_Get() operation on a subscriber process the data necessary to interpret the received block of bytes as an event object needs to be self-contained in the transferred data. That implies that any data inside an event object should be a primitive type (int, float, etc.), a fixed-sized array of primitive types or a fully embedded object. Pointers and references to objects cannot be supported efficiently in this scheme.

```
class Event {
  …
private:
  TypeMap header;
  TypeMap body;
};
```

Figure 4: Embedded TypeMaps in an Event

In the CCA event service, an event consists of two *TypeMaps*, a header and a body, which are mappings of string keys to primitive data types and arrays of primitive data types. We have carefully crafted an implementation of these *TypeMaps* that meets the above criteria with respect to embedded objects inside events. The header and body are fully embedded instances of the *TypeMap* class, each having a fixed, compile-time constant size, while still maintaining the flexibility of supporting a general mapping of string keys to data.

In this manner, we can support the transfer of an event object from publisher to subscriber in a single ARMCI_Get() operation which does not require heavy-duty preprocessing to serialize and reconstruct complex data types. Figure 4 shows the declaration for the *Event* class with its embedded *TypeMaps*. Figure 5 shows a sketch of the code used to read an event on a subscriber process from its remote location on the publisher. It also shows that once an event has been

copied to the subscriber's address space, it can be interpreted as a regular C++ object instance without any preprocessing.

4.1 Memory Management Issues

In our design we wanted to preserve the object-oriented nature of the draft CCA event service specification, while still achieving performance comparable to procedural message passing protocols such as MPI. For this reason, we decided to implement publish and subscribe operations using objects allocated in shareable ARMCI memory areas.

```
char evBuf[SIZE_EVENT];

…

ARMCI_Get(remotePtr,    evBuf,    SIZE_EVENT,
publisher);

Event *ev = reinterpret_cast<Event *>(evBuf);


listener.processEvent(*ev);
```

Figure 5: Accessing an event on a subscriber process

The allocation of shareable ARMCI memory areas is a relatively expensive *collective operation* (ARMCI_Malloc()) in which all processes executing the application must participate. For this reason, we avoid allocating memory through ARMCI_Malloc() every time we need to allocate a shareable object. We allocate a single, large chunk of memory from ARMCI and then use a specialized heap manager for allocate and deallocate sub-chunks from it.

Our heap manager takes advantage of the fact that the sizes of objects that will be shared through ARMCI are known at compile time: events have a fixed size that depends only on the compile-time size of their constituent *TypeMaps*. Other objects that are allocated on the ARMCI heap (list links, etc.), also have known sizes and are much smaller than the event objects.

To simplify heap management and facilitate the creation of classes that allocate objects on the ARMCI heap, we created a base class named ARMCIAllocatable that provides specialized new() and delete() methods that allocate instances on the ARMCI heap. Thus allocating an event object on the publisher that will be accessible to remote subscriber processes becomes simply:

```
Event *ev = new Event;
```

The main limitation that our approach has is that any pointers or references inside an object allocated on the ARMCI heap must be to other objects on the ARMCI heap. Importantly the transfer of these objects to a remote address space (through ARMCI_Get() or ARMCI_Put()) is explicit and thus any pointer links must be followed explicitly. For this reason, our *Event* objects have no links or references within them and have their member *TypeMaps* fully embedded within them.

5. Preliminary Performance Results

In order to test the performance and functionality of our event service implementation, we created a test application that executes the event service on a Cray XD-1 platform on different numbers of processors. The Cray XD-1 system is a distributed memory HPC cluster. On the XD-1, dual-processor AMD Opteron SMP nodes are connected through a high-speed proprietary Cray interconnect named RapidArray [10].

Our test application initializes the event service and then a single publisher proceeds to publish 2500 events of a fixed size. Different numbers of subscribers will then consume those events from the publisher's memory. The consumption of events simply counts the number of entries in the event's *TypeMaps*. The application measures the number of events that can be processed per second for a different number of subscribers.

Figure 6 presents the number of events processed per second for different numbers of subscribers (1 to 16) and for two different event sizes: small (4 KB) and large (50 KB). The number of events processed per second ranges from 66,560 for the small event size on one subscriber to 954 for the large event size on 16 subscribers. The performance drops off as the number of subscribers increases due to contention on the memory subsystem and RapidArray network port for the publisher node.

We used both processors on the Cray XD-1's nodes for executing our test application. It is important to note that for the 50 KB event size the achieved *data rate* (size of events times the number of events/second) using one subscriber is 1006 MB/sec (50 KB * 20995 events/second), which is comparable to the raw MPI bandwidth we measured for the same data size (1200 MB/sec) between two nodes. The event data rate includes the overhead in performing processing of the event, while the MPI data rate includes only the data transfer time.

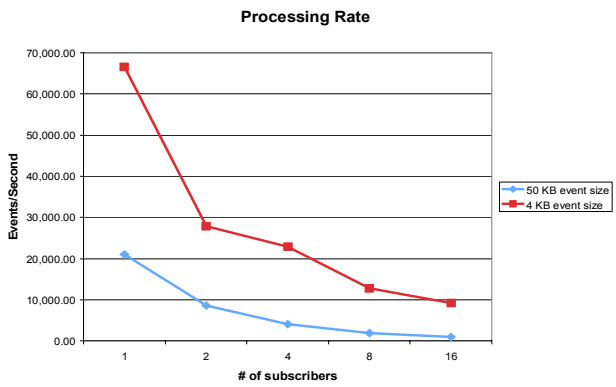

Figure 6: Event processing rate for different numbers of subscribers

These results indicate that our scheme which places the data on the publisher's local memory to be remotely consumed by the subscribers requires a careful balance between the number of subscribers and the capability of the publisher to serve their requests. However, with a small number of concurrent subscribers (<= 2) our object-oriented scheme is able to achieve high performance that approaches that of raw HPC message passing using MPI.

6. Conclusions and Further Work

We have demonstrated the feasibility of implementing a publish-subscribe based event service on an HPC platform, which supports high-level object-oriented interfaces and can achieve high performance comparable to raw message passing using MPI. Our scheme uses a *pull*-based strategy in which subscribers remotely consume events from a publisher's memory which limits its scalability with higher numbers of concurrent subscribers.

As future work, we plan to study alternative publish-subscribe schemes, such as a *push*-based strategy in which the publisher remotely places events onto the consumer's memories, and other possibilities include taking advantage of networking techniques such as multicast to reduce the need for the publisher to individually send data to each subscriber. Another possibility within a *pull*-based approach is to replicate the published data onto a set of nodes such that no single node becomes a bottleneck.

7. References

[1] Y. Ashlad, B. E. Martin, M. Marathe, C. Le. Asynchronous notifications among distributed objects. In *Procs Conf on Object-Oriented Technologies and Systems*. Usenix Association, June 1996.

[2] http://www.omg.org/technology/documents/corbaservices_spec_catalog.htm

[3] http://java.sun.com/products/jms/

[4] P. Tran, J. Gosper, I. Gorton: Evaluating the sustained performance of COTS-based messaging systems. *Softw. Test., Verif. Reliab. 13(*4): 229-240 (2003)

[5] S. Chen, P. Greenfield: QoS Evaluation of JMS: An Empirical Approach. HICSS-37, January 5-8, IEEE, 2004

[6] C.R. Johnson, S.G. Parker, and D.M. Weinstein. "Component-Based Problem Solving Environments for Large-Scale Scientific Computing," Concurrency and Computation: Practice and Experience, 2002 14:1337-1349.

[7] J. Nieplocha, V. Tipparaju, M. Krishnan, D. Panda. High Performance Remote Memory Access Comunications: The ARMCI Approach. Int. J. High Performance Computing and Applications, Vol 20(2), 233-253p, 2006

[8] R. Armstrong, G. Kumfert, L. Curfman McInnes, S. Parker, B. Allan, M. Sottile, T. Epperly, T. Dahlgren. The CCA Component Model for High-Performance Computing. *Concurrency and Computing: Practice and Experience*, 18(2):215--229, 2006

[9] Eisenhauer, G., Bustamante, F. E.,Schwan, K.. Event Services for High Performance Computing. In *Procs 9th IEEE Int. Syp. on High Performance Distributed Computing (Hpdc'00)* (August 01 - 04, 2000).. IEEE

[10] Tripp, J. L., Hanson, A. A., Gokhale, M., and Morveit, H. Partitioning Hardware and Software for Reconfigurable Supercomputing Applications: A Case Study. In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 27, Washington, DC, USA, 2005. IEEE Computer Society.