Realizing Large Scale Distributed Event Style Interactions

A Vijay Srinivas, Raghavendra Koti, A Uday Kumar and D Janakiram
{avs@cs.iitm.ernet.in}, {raghuk, udaya@peacock.iitm.ernet.in}
{djram@cs.iitm.ernet.in}
Distributed & Object Systems Lab,
Dept. of Computer Science & Engg.,
Indian Institute of Technology, Madras, India.
http://lotus.iitm.ac.in

Abstract

Interactions in distributed object middleware that are based on Remote Procedure Call (RPC) or Remote Method Invocation (RMI) are fundamentally synchronous in nature. However, asynchronous interactions are better suited for large scale distributed systems. This paper presents the design and implementation of an asynchronous event based communication paradigm. Typed events, fully distributed hierarchical event dispatching as well as causal delivery of events are the key features that are supported. The paradigm has been realized over Virat, a wide area shared object space that we have built. Virat uses replication, caching and distributed services as the main concepts and provides a well published interface, as well as various relaxed consistency models. This communication abstraction is especially beneficial to applications such as modernizing Airspace Systems, where scalable asynchronous event notification is a critical issue.

1 Introduction

Communication in traditional distributed object middleware such as the Common Object Request Broker Architecture (CORBA) [1] [2] or DCOM [3] are based on Remote Method Invocation (RMI) or Remote Procedure Calls (RPC). These communication styles are synchronous in nature, meaning that the client blocks, waiting for the response from the server after invoking a method. Further, there is tight coupling between client and server. These two factors could have an adverse impact on the scalability of the overall system, especially if there is a chain of calls and the roles of client and server are flexible. Thus, asynchronous, decoupled communication is desirable for large scale distributed systems. An asynchronous communication style that has been addressed in several recent efforts, for instance [4] and [5] is the event style.

Several real world applications such as modernizing Airspace Systems[1] can be modeled based on event style of communication. In the airspace system modernization initiative,

[1] http://www.nas-architecture.faa.gov/cats/

it is being envisaged that every entity in the system such as Air Route Traffic Control Centers (or ARTCCs), Terminal Radar Approach Control Facility (or TRACON), Air Traffic Control Towers (ATCT), different aircrafts, airport gates, automobile rentals, etc. would be connected and events occurring such as arrival of flights, need to be notified to registered entities like airport gates and automobile rentals. Another scenario would be: due to adverse weather conditions, flights cannot take-off or land in a particular airport. This information needs to be communicated to several flights which may be scheduled to land there; to other airports so that they can make arrangements for permitting some of those flights to land; to several ATCTs and TRACONs.

Events can be considered as any *interesting occurrence* for a specified set of *observers*, which are entities interested in being notified when events occur. Events in the system can be mapped external events or changes to the state of certain objects. Observers can register for events, announcing their interest in the specified event. The system needs to asynchronously notify the observers when the event occurs. If the number of observers interested in a given event is high (order of thousands or even more) and there are large number of events occurring in the system, then scalable event notification becomes a critical issue.

This paper presents a new methodology for realizing the event communication paradigm. It presents the design and implementation of a scalable event communication paradigm, which has been realized over Virat. Virat is a large scale object based Distributed Shared Memory (DSM) system that we have built [6]. Key features of Virat include: an efficient object location mechanism based on distributed services, a novel mechanism based on object versioning to handle failure recovery, a data centric Concurrency Control (CC) mechanism to realize relaxed consistency models. Virat has now been converted into a shared event space, meaning that the objects can register for events and be notified asynchronously when the events occur. In this case, the occurrence of an event is equivalent to an object being entered into the shared object space. Observers (in this case objects or processes) interested in the event are asynchronously notified. The shared event space has been implemented and is currently being tested over a wide-area network.

The rest of the paper is organized as follows: Section 3 gives an overview of Virat. Section 4 explains how an event infrastructure has been realized over Virat. Section 5 gives the implementation details of the event communication paradigm. Section 6 compares Virat with existing event based systems in the literature. Section 7 concludes the paper and provides directions for future research.

2 Background: Data Centric Concurrency Control

This section explains the data centric CC that has been proposed earlier by the fourth author and others in [7]. This approach is different from the traditional CC algorithms, in which a process external to the data items keep track of currency information and is also responsible for synchronizing the transactions. The key idea is that meta-data of data items maintain the information about the currency. A data counter is associated with each data item, indicating the number of operations that have been performed on it. There are two types of counters, read and write counters, to keep track of read and write (update) operations. The CC algorithm uses these values to order transaction access to the data item. In the context of three-tier systems, the algorithm works as follows:

Transactions are submitted from the client tier to the application servers. They initially

execute at these servers. The initial phase is a read phase, in which transactions read the data item from the database. The application servers maintain a cache of data items (along with the meta-data information). If the required data items are found in the cache, transaction proceeds without accessing the database server in read phase. Otherwise, the database is contacted for information about specific data items. The next phase is called initial validation phase, in which the transaction has completed its execution and needs to update the database. It presents the newly computed value and the counter values it read earlier to the application server. If the values match, then no other transaction has updated these data items concurrently and this transaction can proceed to the final phase. If the values differ, the transaction is aborted or restarted with new counter values. The last phase is a final validation phase, in which validation (similar to initial validation) is performed at the database server.

3 Overview of Virat: A Large Scale Distributed Shared Object Space

Virat uses a hierarchical organization of distributed object repositories and lookup servers for locating objects in the DSM, with one repository and lookup server for each cluster in the system. Clients can use any shared object knowing only the class name or object identifier. Lookup servers maintain a mapping from class name to object identifiers (oid) in addition to maintaining the current location of the object repository. This provides flexibility to migrate the object repository for load balancing within the cluster and recover from repository failures. Object repository stores information about objects created in that particular cluster. This includes list of accessors for each object, its oid and a copy of the object.

The client code consists of instantiating a DSM runtime object and invoking methods on it. The interface of the DSM runtime object provides methods for creating, reading, and writing shared objects. The DSM runtime object creates a shared object by first invoking a method on the lookup server to check if the object has been created already. If there is no entry in the lookup service, then the DSM runtime object sends a request to the object repository in that cluster. The object repository creates a new entry for the object, if it has not been created in any other cluster. Virat also uses effective caching schemes for the lookup information, by using shared cache objects in each node. These shared objects are updated asynchronously whenever the set of objects it has cached are modified.

Consistency is an important issue in any DSM system, especially if Multiple Readers Multiple Writers (MRMW) model of DSM is used. In Virat, a data centric Concurrency Control (CC) mechanism is used to ensure consistency [7, 8]. Each object has a counter associated with it, representing the number of updates on that object. The counter is used to check for conflicting updates, thus avoiding the use of expensive vector clocks or other schemes for CC. In addition to the CC mechanism, the consistency criteria also plays an important part in DSMs. Realizing strict consistency is expensive, especially in a distributed system. Thus, Virat provides two relaxed consistency models: causal consistency and causal serializability [9]. The applications can choose the desired consistency model and Virat ensures its realization. The object repositories are responsible for propagating updates to the list of accessors for each object.

Virat has been implemented using J2EE platform[2] over a multi-cluster environment: Three different clusters, each with about 10 machines of varying configurations (from P2 to P4 processors, 6MB to 1GB of memory, and so on), having different architectures (SPARC or Intel) and running different operating systems (Linux, Solaris, etc.) The object repositories and lookup servers are implemented as Java Remote Method Invocation (RMI) servers, waiting for requests from its clients (DSM runtime objects). The client code consists of instantiating a DSM runtime object and invoking methods on the DSM runtime object to access shared objects.

4 Event Communication Paradigm over Virat

We provide several key features for the event communication paradigm over Virat: (i) Fully distributed hierarchical event dispatching scheme. (ii) The ability to specify arbitrary event types (iii) Causally ordered delivery of events. The hierarchical event dispatching scheme coupled with the causal event ordering (a relaxed event ordering scheme compared to total ordering [9]) makes the event communication paradigm scalable. The following sections elaborate on some of these ideas.

4.1 Event Registration and Dispatching

Processes or objects can register for events with the local DSM runtime of Virat. The DSM runtime in turn conveys the request to the local object repository of the cluster. If that object repository maintains information about the event (each event has an event identifier (eventID)), it adds the DSM runtime to the list of notifiers for this event. However, if a different object repository maintains information about the particular event, the first object repository sends a request to the second and gets added to list of notifiers at the second repository. Thus, the notifier list for each event can be either DSM runtime entities or object repositories themselves. This results in a hierarchical dispatching of events, from object repositories to other repositories and then to DSM runtime and then finally to their respective objects/processes.

4.2 Event Ordering

Delivering events in a totally ordered fashion is a difficult task in distributed systems. Racing messages and message failures can cause ordering algorithms to fail. Further, a naive event ordering scheme, for instance, one based on [10], would hinder the scalability of the event paradigm, with message overheads and waiting times proving to be bottlenecks. Thus, a sophisticated event ordering scheme that would also be easier to implement is required. Event counters are used to realize causally ordered event delivery, similar to the use of data counters for CC in Virat. Thus, if an event affects another event causally, then they are delivered in the same order across all processes/objects in the system. There are no explicit guarantees about the delivery for events not causally affecting each other. The event counter based ordering scheme is expected to perform better and scale up compared to approaches based on causally ordered message delivery [11].

[2]It can be implemented just as easily over any other platform such as CORBA or even .NET, using any language, say $C++/C\#/VB$.

4.3 Event Types

An instance of a class that inherits a standard class called *Counter* can be an event object. This makes the event object serializable and is also useful in realizing event ordering. Thus, event types can be any user defined type. This is a simple and elegant mechanism to realize typed events. The occurrence of an event indicated by inserting an event object into the DSM using the *putEventObject* method provided by the DSM runtime object. This would result in the event object being sent to all the notifiers.

4.4 Events over DSM

Existing event infrastructures and service/component composition mechanisms have been realized over middleware or lower level network support, not over a shared object space. This has several advantages: Application or even object specific policies can be enforced; object state changes can be mapped to events and interested observers notified; object interface changes (evolution) can also be mapped to events and clients notified; special Quality of Service (QoS) policies can be enforced; events can also be composed, just as components/objects are composed currently in Virat.

5 Implementation of Event Communication Paradigm

Distributed event style mechanism separates event generation and event dispatch with the help of the wide area distributed shared memory (Virat) explained in section 3. This clear separation ensures that entities distributed over a wide area network can signal occurrence of events. They can also act as observers for other events and be notified when the events occur. Observers are allowed to register for the events of their choice. Thus, the asynchrony facilitates a scalable distributed shared events space to be built over a DSM. Implementation of the event style involves mainly three parts: the event registration, the event generation and the event dispatching parts. Each of them is briefly described below.

Event registration enables a process to register for event notification through the API *int registerForEvent(String eventType, ...)*. The identity returned is used to identify the process so that future event notification can be delivered to it. Event generation is enabled by the method *putEventObject(Counter eventObj, type eventType, ...)* which puts an event object into the distributed shared memory. This is all that the event generating software needs to do, for event signalling.

The event dispatch mechanism recognizes that an event object is placed. It communicates the event to all the registered DSM runtime objects by invoking the *notify* method. If there is no entry for this particular eventType, other repositories are contacted and the event object is forwarded to those repositories which have an entry corresponding to the eventType. The DSM runtime objects forward the event object to the process that have registered for that eventType. Being an asynchronous event delivery mechanism, the processes are expected to check for the event using the interface *checkForEvent(int index, String eventType, ...)* where index is the value returned by the method call *registerForEvent*. The event object is returned to the process which calls CheckForEvent so that further action can be taken based on the state of the event Object. A null is returned if the event has not yet occurred between two invocations of CheckForEvent call. Callbacks can be used as an alternative to polling. During registration, the processes

provide a callback object in addition to the event type they are interested in. Once the event occurs and the DSM runtime is notified, it in turn notifies the callback object. The hierarchical dispatch mechanism is illustrated in Figure 1.

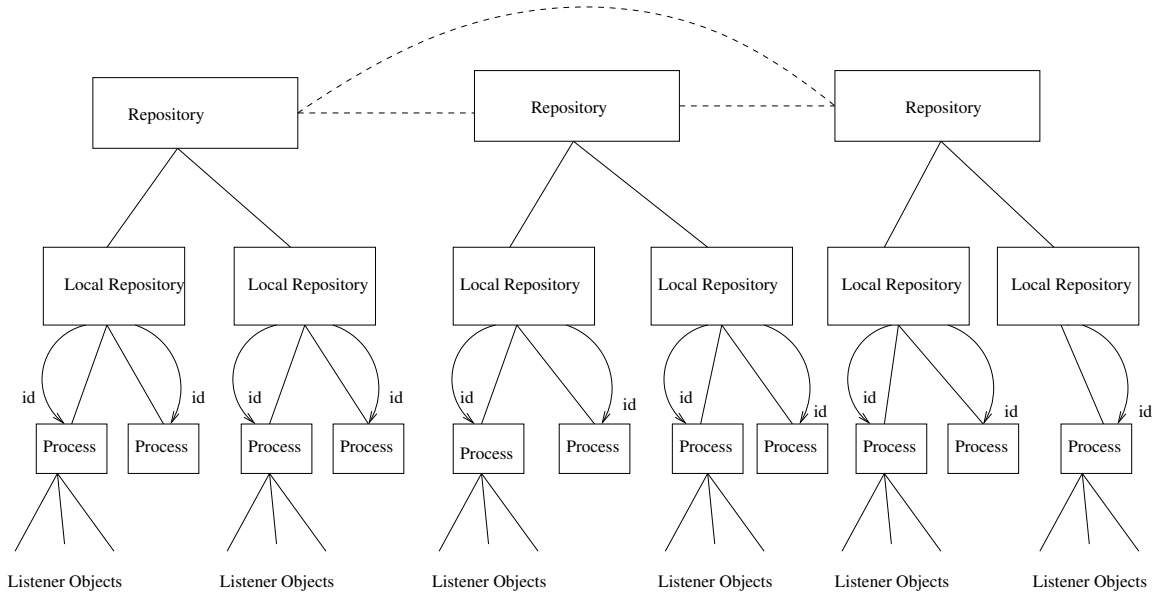

Figure 1: Hierarchical Event Style Infrastructure

6 Related Work

The event paradigm as such is not a new idea and has been used by several software systems. CORBA has a sophisticated Notification Service [12], an improvement to the OMG Event Service that existed earlier. It specifies the IDLs for event suppliers, which supply event messages, event consumers, which consume event messages and event channels, which is like an event dispatcher. It also contains mechanisms for event filtering to allow clients to specify exactly which events they are interested in. Further, events can be typed or untyped events. However, there is a gap between the specification and actual implementations. Current implementations only provide untyped events and centralized event dispatching mechanisms or direct connection between event supplier and consumer. In contrast, Virat[3] provides typed events and completely distributed event dispatching mechanisms.

JavaBeans provides a simplified event model [13]. Suppliers of events need to be contacted by event consumers directly, as JavaBeans does not explicitly support an event dispatcher. However, different event types are supported by subclassing the basic class called EventObject. In Virat also, a similar approach is followed for handling typed events. Moreover, Virat supports a distributed dispatching mechanism, that leads to true decoupling between sender and receiver of events.

Quality of Service (QoS) for event delivery has been explored in detail in [14]. It presents an event stream interpretation language which facilitates consumers to specify

[3]In this section, the term Virat refers to both the DSM and the shared event space over Virat

tolerance for event delivery. This provides the middleware with flexibility to use efficient protocols for event dispatching. However, they do not provide scalable event ordering schemes. Virat provides delivery guarantees for causal events, meaning that event delivery respects causal ordering, not total ordering.

The work closest to Virat in terms of overall approach is Java Event-based Distributed Infrastructure (JEDI) [15]. JEDI provides an object oriented infrastructure to enable the development of event-based applications. The paper [15] also presents a sample distributed work-flow management system implemented on top of JEDI. JEDI provides algorithms for event dispatching, both centralized and fully distributed versions. Similar to Virat, JEDI also provides causal delivery of events. However, the mechanism used to achieve event ordering in JEDI has not been detailed in the paper [15]. The usual mechanisms to achieve causal message delievery are based on vector clocks and hence, may have difficulty in scaling up. Whereas, Virat uses an elegant, scalable data centric approach to achieve causal event ordering. The other difference between Virat and JEDI is in the event types. JEDI provides simple, untyped events only, whereas Virat provides sophisticated event types, even application specific or user defined event types.

It can be seen that the combination of features provided by Virat, namely, supporting typed events, fully distributed event dispatching mechanisms and causal delivery of events makes it a unique event system. Further, scalability has been one of the main design goals of Virat. Virat is expected to scale both in terms of the number of event consumers and the number of different events in the system simultaneously. Further, the shared event space is realized over a wide area shared object space. This has specific advantages, as outlined in section 4.4.

7 Conclusions

This paper has presented the design and implementation of a distributed event based communication paradigm over Virat, a wide area DSM. Key aspects of the event space are the fully distributed hierarchical event dispatching scheme, causally ordered event delivery and typed events. Scalability has been one of the main design goals of this system. This work would be very useful in a number of applications that need large scale event notification, both in terms of number of observers and number of events. The National Airspace System (NAS) system modernization effort is a perfect fit, as NAS can be modeled as a large scale event based distributed system. An interesting future direction would be to extend Virat to distributed mobile systems and investigate whether efficient shared object and shared event abstractions can be realized over it.

References

[1] Object Management Group. The Common Object Request Broker: Architecture and Specification. 2. 3. 1, October 1999.

[2] Michi Henning and Steve Vinoski. *Advanced CORBA Programming with C++.* Addison Wesley, Massachusetts, 1999.

[3] G Eddon and H Eddon. *Inside Distributed COM.* Microsoft Press, Redmond, Washington, 1998.

[4] Umar Saif and David J Greaves. Communication Primitives for Ubiquitous Systems or RPC Considered Harmful . In *Proceedings of ICDCS International Workshop on Smart Appliances and Wearable Computing*. Phoenix (Mesa), Arizona, USA, April 2001.

[5] Jean Bacon *et al.* Generic Support for Distributed Applications. *IEEE Computer*, 33(3):68–76, March 2000.

[6] A Vijay Srinivas, D Janakiram, and Raghevendra Koti. Virat: An Internet Scale Distributed Shared Memory System. Technical Report IITM-CSE-DOS-2004-03, Distributed & Object Systems Lab, Indian Institute of Technology, Madras, January 2004.

[7] D Janaki Ram, N S K Chandra Sekhar, and M Uma Mahesh. A Data-Centric Concurrency Control Mechanism for Three Tier Systems . In *Proceedings of the IEEE Symposium on Web caching for E-business, part of IEEE Conference on Systems, Man and Cybernetics*. Arizona, USA, 2001.

[8] D. Janaki Ram, M. Uma Mahesh, N.S.K. Chandra Sekhar, and Chitra Babu. Causal Consistency In Mobile Environment. *ACM Operating Systems Review*, 35(1):34–40, January 2001.

[9] M Raynal, G Rhia-kime, and M Ahamad. Serializable to Causal Transactions for Collaborative Applications. In *Proceedings of the 23rd Euromicro Conference*. Budapest, Hungary, September 1997.

[10] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):124–149, July 1978.

[11] M Raynal and A Schiper and S Toueg. Causal Ordering Abstraction and a Simple Way to Implement It. *Information Processing Letters*, 39:343–350, 1991.

[12] Object Management Group. Notification Service Specification. 1. 0. 1, August 2002. formal/02-08-04.

[13] Sun Microsystems. JavaBeans. 1. 01, August 1997.

[14] Yuanyuan Zhao and Rob Strom. Exploiting Event Stream Interpretation in Publish-Subscribe Systems . In *Proceedings of ACM International Conference on Principles of Distributed Computing (PODC 01)*. Newport, Rhode Island, USA , August 2001.

[15] Gianpaolo Cugola and Elisabetta Di Nitto and Alfonso Fuggetta. The JEDI Event-Based Infrastructure and Its Application to the Development of the OPSS WFMS. *IEEE Transactions on Software Engineering*, 9(27):827–850, September 2001.