Active Brokers and Their Runtime Deployment in the ECho/JECho Distributed Event Systems

Dong Zhou, Yuan Chen, Greg Eisenhauer and Karsten Schwan
College of Computing, Georgia Institute of Technology
{zhou, yuanchen, eisen, schwan}@cc.gatech.edu

Abstract

This paper introduces active brokers and the third-party derivation, the basic programming construct for runtime remote broker deployment, in the ECho/JECho distributed event systems. We describe its implementation in the JECho system and give examples of using it in ECho/JECho distributed event systems. In particular, we describe the use of third-party derivation in supporting the scalability of content-based event delivery. Specifically, third party derivation is used both to dynamically construct content-based event distribution trees and to offload potentially expensive client-specific event routing/processing by runtime creation of remote brokers. Our preliminary benchmark results demonstrate significant benefits of using third-party derivation.

1. Introduction

Event selection and conversion in publish/subscribe systems. Large scale distributed publish/subscribe systems depend on event matching or filtering to deliver events solely to interested clients. Examples of event matching or filtering include their selection based on type[3][2], their selection based on application-specific headers that may include client identifiers[4][3][5][6][7], and their selection based on arbitrary elements of their data content[3][5]. Event conversion or transformation performed prior to their receipt by clients is a further step toward customizing event delivery to individual clients' needs.

Out of client event processing. There are various reasons for event conversion and transformation to happen outside of clients, before the events reach individual clients. Performance enhancement is the most obvious one. Specifically, performance enhancement is possible because such conversion can potentially:

- *Minimize network traffic:* event conversion may result in new events of smaller sizes.
- *Offload computation from slower or power-constrained clients:* for clients with limited computational resources or limited power supply (e.g., PDAs), offloading computation from clients to brokers can either speedup event delivery or reduce power consumption. An example is for event brokers to pre-convert events to the forms needed by specific client's graphical displays [7][6][5], saving the cost of format conversion in the client.
- *Enable event filtering:* in some applications we have developed, filtering to reduce system traffic is possible only after certain conversions are performed. For example, consider data events that carry scientific data to be distributed across multiple collaborating scientists viewing this data[10]. Often, such distribution follows rules that may be applied only after event data has been transformed, as with atmospheric data that is routed to collaborating scientists according to its temporal and spatial distribution across the Earth's grid. This distribution is possible only after the data has been transformed from its spectral to its grid form[8].
- *Streamline event transport:* events sent between heterogeneous machines can be converted to client machine formats before being sent out to avoid the overheads in using standard intermediate formats, yet doesn't sacrifice ease of use if it is done transparently by middleware[11].

In addition, individual applications may have more specific reasons for doing out-of-client event conversions. Such conversion could be vital for the correctness of some applications and be important to the quality of service of some others. We will describe some of such applications in Section 3.2.

Problem statement: efficient event distribution and out of client event processing. A well-known problem in distributed event systems is the trade-off between the latency of event distribution vs. scalability with respect to the distribution of events to large numbers of clients. Low latency suggests the use of point to point connections between event producers and consumers[3][5]. High scalability suggests the use of distribution trees and event routing[4]. Out of client event processing complicates this trade-off, since it may add processing load to event producers, but it may also filter events, thus reducing the network bandwidth needed between event producers and consumers. Further, the actual processing loads vs. bandwidth needs implied by out of client processing are difficult to assess before runtime.

Supporting out of client processing with active brokers. (J)ECho supports a special kind service objects called *active brokers*. An active broker is a special service object. In addition to normal service requests, an active broker can accept program logic from clients. The broker provides an environment for client submitted program logic to be executed at certain moments. The behavior of an active broker is jointly defined by the logic of the broker itself and the logic submitted by the clients. An example is a content-based event router, whose routing algorithm is partially defined by its clients' filter functions. Active brokers differ from conventional event brokers in that (1) they are deployed at runtime, and (2) they may be deployed into event consumers, producers, or into separate processes. This permits event systems to be dynamically configured to match the event traffic they are carrying and their clients' current needs. For instance, low latency of event distribution between producers and consumers may be achieved by placing active brokers into producers and creating point to point connections between producers and consumers[13]. Scalability to large numbers of clients may be attained by creating additional processes into which active brokers are deployed, and by performing event routing across multiple event brokers, thus creating at runtime suitable event distribution trees[4].

Third Party Derivation. In the (J)ECho systems, dynamic active broke deployment is supported by *Third Party Derivation* (3PD), a novel event system-level programming construct to permit system developers and applications to deal with the overheads of event processing and distribution in publish/subscribe middleware. Derivation creates a new stream of events by client-controlled filtering and/or processing of an original stream of events. Third party derivation means such derivation is carried out at remote sites. Since derivation happens at runtime, an important issue in 3PD is to ensure clients' event observation consistency before and after derivation.

In the remainder of this paper, we first introduce active brokers in the context of JECho implementation. We then explain the notion of third party derivation in more detail. This explanation is followed by a description of how 3PD will be and is being used in the (J)ECho systems. Microbenchmarks evaluating the effects of 3PD appear last.

2. Active Brokers in the JECho Distributed Event Systems

JECho[17] is a Java-based efficient distributed event system. JECho offers the abstractions of events and event channels. An event is an asynchronous occurrence, and may be used both to transport data and for control. An event endpoint is either a producer that raises an event, or a consumer that observes an event. An event channel is a logical construct that links some number of endpoints to each other. An event generated by a producer and placed onto a channel will be observed by all of the consumers attached to the channel. Each event consumer can submit an event filter, which can include complex computtaion, to the system. The system puts the filter into an active broker, which can reside at the data source, in the client's address space or in other remote process.

Active brokers provides an enviroment for event filters to (1) permit the filter to carry out computation with necessary access to resources and (2) notify the filter at significant moments (e.g., when an event has been emitted or when network bandwidth is available). In addition, since a filter can be replicated when there are multiple data sources, it is necessary for the environment to provide facilities for maintaining state consistency across filter replicas whenever necessary. In response to these requirements, active brokers provide three interfaces to filters:

- the *resource control interface* exports and controls 'capabilities' based on which filterss can access system- and application-level resources;
- the *shared object interface* provides consistency control for replicated filterss that share state; and
- the *intercept interface* defines a set of functions that can be provided by a filters and later be invoked at certain significant moments.

3. Runtime Remote Broker Deployment with Third-party Derivation

The presence of many filters in an active broker may cause the degradation of its performance. (J)ECho handles this problem by dynamically integrating new active brokers into the event distribution system that is currently running. This runtime deployment of active brokers is supported by a novel programming construct, termed *Third Party Derivation* (3PD).

3.1 Definition of Third Party Derivation

Third-party Derivation (3PD) is a mechanism for runtime remote proxy deployment in distributed event environments. 3PD takes as input one or more original channels,

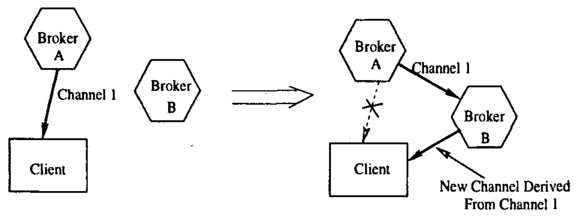

FIGURE 1. Illustration of Third Party Derivation

```
Class MySubscription extends  FIFOFilter{
    void enqueue(Event e) {
        Stock s = (Stock) e.getContent( );
        if(s.name == "IBM")
          super.enqueue(e);
        else if(s.price > 100)
          super.enqueue(e);
        return;
  }
```

FIGURE 2. Sample Subscription Code

the name and location(s) of the desired remote service, and returns a new channel. 3PD may be explained in two steps, the first concerning the basic notion of *deriving* an event channel, the second concerning the involvement of a *third party* in such a derivation. Intuitively, whenever a client wishes to specialize an existing event channel in order to filter or select the events being received differently, or to change how they are processed, the client *derives* the event channel from its current to some new form (Figure 1). Part of this derivation is the client's provision of a new 'filter' function that must be applied to the event before the client receives it. This filter function is applied to all events received from the new channel that has thus been 'derived' from the existing one. Clients that use the derived channel receive the filtered events. Clients that continue to subscribe to the original channel are not affected. In this fashion, at runtime, (J)ECho can create from some event source any number of additional events of interest to clients, selecting them based on type, on client id, or on event content.

Event channel derivation must guarantee certain correctness rules, in order to maintain consistent behavior as perceived by event consumers. Assume that Co is the old channel and Cn is the newly derived channel. Events Eo(0), Eo(1), Eo(2)...Eo(t), Eo(t+1)... describe the event stream in the channel Co and events En(0), En(1), En(2)... describe the event stream in Cn. Further assume that event En(0) is derived from event Eo(t), then it must hold true that this particular client should never observe events Eo(i) for i>=t, and that this same client should have observed events Eo(i) once and only once for i<t. In addition, All Eo(i) observed by the client should be observed before any En(j).

For example, suppose we have a visualization client receiving and displaying a uncompressed video stream. If we wish to deploy, at runtime, an active broker that compresses image frames before they are sent to the client, this implies that the client must also uncompress the frames it receives. The consistency constraint defined for channel derivation and stated above says that, assuming the broker starts compression at frame 20, the client should receive frames 0-19 in uncompressed form and frames 20 and beyond in compressed form. Further, frames are correctly ordered and should be uncompressed starting at frame 20.

This seemingly simple constraint complicates the implementation of runtime channel derivation, considering the typically asynchronous nature of events, the possible presence of multiple event providers, and the fact that the broker may be running in a process separate from the event consumers or producers.

3.2 Using Third-party Derivation

3.2.1 An Example of Consumer-Initiated 3PD

The introduction has already mentioned several examples of consumer-based 3PD. We add to these examples a simple scenario we are currently implementing for the wireless domain, where a single multi-media data server is supplying data to a large number of wireless clients. Specifically, consider a sports stadium in which a large number of users receive video data from a central server, which was first captured by cameras offering different views of the field and players, and is then distributed to clients as per their current interests in players, different views of the game, etc. In this case, a large number of clients will require the central data server to perform considerable data selection and routing actions, an issue we address by use of 3PD. That is, in response to client-initiated channel derivation to serve individual clients' needs, the event system can use 3PD to create third party proxies on other nodes of the cluster- or SMP-based data server, thereby dealing at runtime with the dynamic behavior exhibited by clients. Note that certain stadiums already offer seat-resident PDAs capable of actions like these to limited degrees and that companies like SONY are developing interactive video games in which players can 'race' their cars in real-time against those driven by actual NASCAR race drivers[15].

Since (J)ECho now runs across wireless laptop- and PDA-based platforms, we are experimenting with scenarios like these, where it is desirable to reduce the use of wireless media shared by a large number of clients, while at the same time, increasing data communication and processing in the tightly coupled cluster servers that provide processed and/or down-sampled data to clients[16].

3.2.2 An Example of Supplier-Initiated 3PD

While previous text has indicated the utility of 3PD from the perspective of event clients, an equally important benefit is the use of 3PD by an event supplier, in order to dynamically deploy new proxies for improving event delivery. For example, assume that an embedded device (e.g., a camera or infrared sensor with a built-in communication processor and board) is sending out data via an event channel, and that this device is connected to its host via a wireless network. Instead of using multiple point-to-point connections from this smart device to every consumer of its data (e.g., multiple users viewing the same camera data or requiring the infrared sensor's output), in this scenario, it is better to use 3PD to construct at runtime, a third party that first receives events from the wireless sensor and then distributes them (via wired links) to all interested clients. A similar example occurs in the case of large-scale operational information systems, like those we have been investigating jointly with collaborators from Delta Air Lines[12]. In Delta's wide-area system, for example, each of its airports is connected to its centralized data processing system (located in Atlanta) via leased lines. Line costs depend on the total bandwidth guaranteed by the carrier, so that it is preferable to concentrate events within each airport LAN, thus performing all possible event filtering and concentration there. In JECho, such concentration is achieved by deploying into the airport active broker process, using 3PD, which performs all necessary event concentration and filtering, and then forwarding only the events needed by the central side via the leased lines. Unfortunately, such 'smart' event filtering and concentration is not currently done within Delta's OIS, with the exception of using TIBCO's LAN/TCP-based 'multicast' solution to reduce the total traffic imposed on links to airports.

3.3 Implementing Third-Party Derivation

Java's build-in object and class transport facilities are used to implement 3PD in JECho, the Java version of our system. In this section, we will focus on our algorithm for ensuring event consistency in 3PD. We will use 3PDs initiated by event consumers as examples.

3PD must maintain event order and consistency, as defined in Section 3.1, in the presence of asynchronous event arrivals, for multiple event providers, and with brokers that reside in processes other than event producers or consumers. To attain these correctness properties, we adopt a two-phase algorithm for consumer initiated 3PDs: the first stage is the setup phase, and second stage performs the actual channel derivation. During the setup phase:

1. the consumer initiates 3PD by sending a *3PD request*, which includes the ID of the consumer and the name and destination of the remote service, to all of the suppliers of the channel;
2. upon receiving a 3PD request, a supplier atomically replaces the consumer from its client list with the 3PD destination, sends an *ACK* to the consumer and a *3PD mark* to the 3PD destination;
3. the consumer buffers/processes events from a supplier until it receive the ACK from the supplier; it enters the second stage of the algorithm after it receives ACKs from all the suppliers.
4. upon receiving a 3PD mark, the destination will setup a delegate for the requested service, if no delegate for the 3PD has already been set up; the delegate will buffer events for the intended for the newly deployed active broker.

The actual derivation phase starts when the consumer receives ACKs from all the suppliers. It will then send an *ACTUAL 3PD* request, along with any buffered events, to the 3PD destination. Upon receiving a ACTUAL 3PD request, the destination merges events locally buffered events with events sent along with the request, and then initiates the active broker with merged events.

This algorithm ensures event consistency. It does not disrupt other clients of the channel, and it is relatively efficient.

4. Supporting Scalable Content-Based Event Routing

This section describes content-based event routers in (J)ECho event systems and their support of scalable event routing. It starts with a description of event routers, followed by the use of 3PD for router load-balancing, ending with a brief introduction to the notion predicate extraction and propagation we are pursuing in order to dynamically optimize content-based event routing.

Content-based event routers are the most commonly used active brokers. in (J)ECho. (J)ECho routers allow complex computations to be performed in filter functions, and filter functions can maintain internal states. As discussed earlier, content-based routing may overload routers, because there are too many clients or because the processing loads implied by clients are too high. To address this problem, third-party derivation can transparently deploy active brokers into additional router processes, thereby dynamically

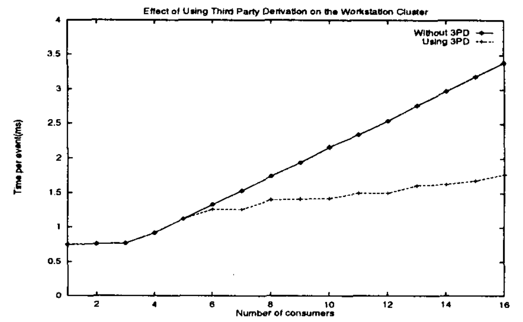

FIGURE 3. Effect of Using 3PD to Build Event Distribution Tree

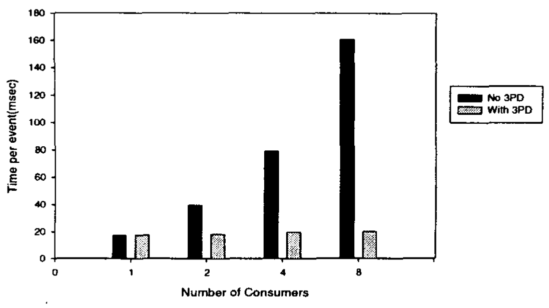

FIGURE 4. Effect of placement of 3PD in a Sample Wireless Setup

constructing a hierarchical routing tree like the one used in systems like Gryphon[1]. For instance, a newly deployed remote router may alleviate the load experienced by an existing router (indicated by a growing number of buffered events), by splitting the existing router's load into two equal parts.

When constructing event distribution trees, it is well-known that in order to support a large number of event suppliers and consumers, it is important to reduce the total number of events being distributed. Ideally, an event should be delivered to only those routers that have consumers interested in the event. Simply moving the filter/processing functions from a lower to a higher level routers can reduce network traffic, but it may also overload higher level routers, thus degrading performance. To solve this problem, we are currently investigating an approach that automatically extracts SQL-like predicate expressions from a user's general content-based subscription function. We then use the predicate subscription information to build a efficient multicast protocol for content-based event routing. We briefly outline this extraction procedure below.

In JECho, a consumer provides its subscription class when it subscribes to a channel. A subscription function (i.e., filter function) performs selection and/or transformation on events. An event is inserted into the outgoing queue by calling method enqueue() in the subscription class. Sample subscription in Figure 2 specifies that the consumer is interested in IBM's stock and in stocks with prices higher than 100.

The extraction procedure initiates its analysis by loading the consumer's subscription class. It builds the control flow graph for the method enqueue(). The analysis proceeds as follows: branch predicates encountered along each of the paths from the entry point to each enqueue statement are recorded. All predicates on the same path are conjoined to form a path condition for that path. All path conditions are disconjoined to attain the final predicate expression that specifies the consumer's interests. The resulting predicate is a conditional expression on the event data and constants. The extraction procedure would discard those predicates that contain either variables or changed event data. The predicate extracted from the above example subscription class would be "name = 'IBM' OR price > 100". We are using SOOT, a Java Bytecode optimization toolkit, to implement the extraction algorithm[18]. With our implementation, extraction could be performed off-line or on-line, when a consumer first derives a channel.

5. Some Benchmark Results

This section presents benchmark results using the JECho implementation. Measurements presented here are performed on two setups. One uses a cluster of 300MHz Pentium II Linux PC connected with dual 100Mbps fast Ethernet. The other uses the same PC cluster plus one iPAQ H3650 handheld device. The iPAQ is connected to the PC cluster via a IEEE 802.11b wireless network.

Building Event Distribution Tree with 3PD. Figure 3 compares the performances of event distribution with and without using 3PD to build a simple event distribution tree in the wired PC cluster. The events being passed are arrays of 100 floats. Without 3PD, all consumers receive events directly from the single supplier. With 3PD, two intermediate brokers are added after 6 and more consumers have joined in the channel. The brokers receive and forward events from the supplier to consumers. Each consumer receives a event either directly from the supplier or from the brokers. A simple load balancing algorithm is used to equally distribute consumers among the supplier and brokers. The figure clearly shows that the rate of event distribution is improved substantially when active brokers are placed into the PC cluster, once the number of clients exceeds a certain limit.

Using Supplier-Initiated 3PD in Certain Wireless Environment. Figure 4 compares the average event delivery time when using or not using supplier-initiated 3PD in a particular wireless setup. Here the wireless iPAQ is the producer of events. Supplier-initiated 3PD is adopted, so that the iPAQ can dynamically choose a single remote broker to which it sends all of its events. The idea is to minimize the use of the wireless communication link. The broker then propagates events to other clients, effectively reducing wireless transmission to once per event, regardless of the number of event subscribers. Figure 4 shows the obvious benefit of using this supplier-initiated 3PD.

6. Conclusions and Future Work

This paper introduces a novel programming construct for event systems, termed Third Party Derivation (3PD). 3PD permits the (J)ECho event systems that support it to react to dynamic changes in the loads implied by event distribution and processing due to runtime client arrivals, departures, and changes in client needs. Essentially, such reactions consist of deploying at runtime, active brokers into the address spaces of event consumers, producers, and into third party processes acting as event routers and processors. This paper evaluates 3PD for the Java implementation of the ECho event system, termed JECho, with both wired and wireless clients. Our preliminary results demonstrate that 3PD can provide essential help for the efficient delivery of events in both environments.

Our future work addresses the automatic deployment of active brokers for both wired and wireless event capture, distribution, and processing platforms.

References

[1] M. Aguilera, R. E. Strom, D. C. Sturman, M. Astery and T. D. Chandra, Matching Events in a Content-based Subscription System, in the Proceedings of Principles of Distributed Computing(PODC), 1999.

[2] P. Th. Eugster and R. Guerraoui, Content-Based Publish/Subscribe with Structural Reflection, in the Proceedings of 6th USENIX Conference on Object-Oriented Technologies and Systems (COOTS'01), San Antonio, Texas, Jan. 2001.

[3] G. Eisenhauer, F. Bustamente and K. Schwan, Event Services for High Performance Computing, in the Proceedings of High Performance Distributed Computing (HPDC-2000), 2000.

[4] G. Banavar, M. Kaplan, K. Shaw, R. R. Strom, D. C. Sturman and W. Tao, Information Flow Based Event Distribution Middleware, in the Proceedings of Middleware Workshop at the International Conference on Distributed Computing Systems, 1999.

[5] D. Zhou, K. Schwan, G. Eisenhauer and Y. Chen, JECho - Interactive High Performance Computing with Java Event Channels, in the Proceedings of the 2001 International Parallel and Distributed Processing Symposium (IPDPS '01), April 2001.

[6] C. Isert and K. Schwan, ACDS: Adapting Computational Data Streams for High Performance, Proceedings of the 2000 International Parallel and Distributed Processing Symposium (IPDPS '00), May 2000.

[7] B. Plale and K. Schwan, dQUOB: Efficient Queries for Reducing End-to-End Latency in Large Data Streams, in the Proceedings of High Performance Distributed Computing (HPDC-9), Aug. 1999.

[8] T. Kindler, K. Schwan, D. Silva, M. Trauner and F. Alyea, Parallelization of Spectral Models for Atmospheric Transport Processes, Concurrency: Practice and Experience, Nov. 1996.

[9] D. Zhou and K. Schwan, Adaptation and Specialization for High Performance Mobile Agents, in the Proceedings of 5th USENIX Conference on Object-Oriented Technologies and Systems (COOTS '99), May 1999.

[10] B. Plale, V. Elling, G. Eisenhauer, K. Schwan, D. King and V. Martin, Realizing Distributed Computational Laboratories, The International Journal of Parallel and Distributed Systems and Networks, Vol.2 No. 3, 1999.

[11] F. Bustamante, G. Eisenhauer, K. Schwan and P. Widener, Efficient Wire Formats for High Performance Computing, in the Proceedings of the ACM Supercomputing 2000 (SC 2000), Nov. 2000.

[12] V. Oleson, Karsten Schwan, Greg Eisenhauer, Beth Plale, Calton Pu and Dick Amin, Operational Information Systems - An Example from the Airline Industry, in the Proceedings of the First Workshop on Industrial Experiences with Systems Software (WIESS), 2000.

[13] G. Eisenhauer, F. E. Bustamante and K. Schwan, A Middleware Toolkit for Client-Initiated Service Specialization, in the Proceedings of the Principles of Distributed Computing Middleware Symposium, Jul. 2000.

[14] D. Zhou and K. Schwan, Eager Handlers - Communication Optimization in Java-based Distributed Applications with Reconfigurable Fine-grained Code Migration, in the Proceedings of the 3rd International Workshop on Java for Parallel and Distributed Computing, Apr. 2001.

[15] SONY Distributed Systems Lab, Interaction with TV in 2003 (Invited Talk), Proceedings of the First Workshop on Industrial Experiences with Systems Software (WIESS), 2000.

[16] R. Gummadi and R. H. Katz, The Data Management Problem in Post-PC Devices and a Solution, in the Proceedings of the SIGOPS EW'00: 9th ACM SIGOPS European Workshop "Beyond the PC: New Challenges for the Operating System", Sep. 2000.

[17] JECho Distributed Event System, http://www.cc.gatech.edu/systems/projects/JECho/.

[18] Soot: a Java Optimization Framework, http://www.sable.mcgill.ca/soot/.