Causeway: Operating System Support For Controlling And Analyzing The Execution Of Distributed Programs

Anupam Chanda, Khaled Elmeleegy, and Alan L. Cox
Department of Computer Science
Rice University, Houston, Texas 77005, USA
{anupamc,kdiaa,alc}@cs.rice.edu

Willy Zwaenepoel
School of Computer and Communication Sciences
EPFL, Lausanne, Switzerland
willy.zwaenepoel@epfl.ch

Abstract

In this paper we introduce Causeway, operating system support facilitating the development of meta-applications, like priority scheduling and performance debugging, that control and analyze the execution of distributed programs. Meta-applications use Causeway to inject and access metadata on application execution paths to implement their specific goals. Causeway has two components: (1) interfaces to inject and access metadata and (2) mechanisms to automate propagation of metadata. Using Causeway we could rapidly implement a distributed priority scheduling system where priority of a task is injected and propagated as metadata, and accessed to implement global priority scheduling. This required writing only about 150 lines of code on top of Causeway. With this system we obtained global priority scheduling on an implementation of the TPC-W benchmark.

1 Introduction

In this paper we introduce Causeway, operating system support facilitating the development of *meta-applications* that control and analyze the execution of distributed programs. Priority scheduling and performance debugging are examples of such meta-applications. A meta-application can span across the application and the operating system (kernel and libraries). Meta-applications use Causeway to inject and access *metadata* on application execution paths to implement their specific goals, e.g., scheduling or debugging. Causeway performs automatic propagation of injected metadata along application execution paths enabling the meta-application to access metadata from anywhere along those paths.

Causeway has two components: (1) interfaces for *actors* to inject and access metadata and (2) mechanisms to automate propagation of metadata to and from actors across *channels*. An actor is an execution context; it can be a process, a thread (in a multithreaded program) or an event handler (in an event-driven program), whether executing in user or kernel mode. Application execution is performed by one or more actors. An actor may communicate with other actors during an execution. A channel is defined as the means of communication between two (or more) actors. Metadata is arbitrary data that is distinct from application data but is propagated alongside application data through the execution paths of the distributed program. Causeway interfaces can be called from both the application and the operating system. Causeway automatically propagates metadata between actors across channels without the need for any application modification.

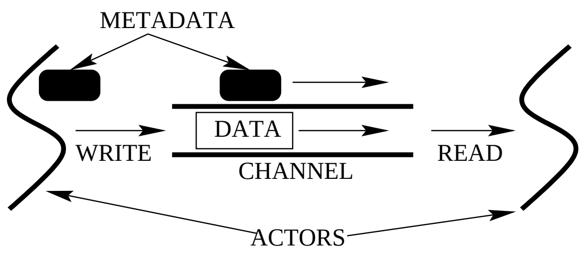

Figure 1: Propagation of Metadata Between Two Actors Across a Channel

At an abstract level, Causeway works as follows. Metadata is associated with an actor when that actor performs injection. Later, when the actor writes application data to a channel, its metadata is associated with the application data written. On a subsequent read from the channel by either the same or a different actor, the metadata is propagated to the actor performing the read. Figure 1 illustrates the concept of propagation of metadata between two actors across a channel.

The complete set of channel types are: (1) sockets, (2) pipes, (3) files, and (4) shared memory. Causeway propagates metadata along a channel on read and write operations by an actor. Some of these channel types are visible to the operating system (kernel and libraries) while others are not. Pipes, sockets and files are system visible whereas shared memory is not. Further, some channel types are persistent, e.g., files, while others, like shared memory, are short-lived. Causeway currently propagates metadata across socket and pipe channels. As ongoing work we are adding support in Causeway for file and shared memory channels.

There are quite a few challenges in the design of Causeway. First, when metadata is propagated to an actor, a decision needs to be made about what to do with the existing metadata on the actor. It is possible that the incoming metadata pertains to a new request to the system: in this case the incoming metadata needs to be assigned to the actor which loses its existing metadata. Alternatively, the incoming metadata may be associated with the same request as being currently executed by the actor but carry a different value: in this case some composition of the incoming metadata and the existing metadata needs to be applied to the actor. Second, on a read on a channel, different pieces of data may be associated with different metadata. Again, a decision is required about what metadata to propagate to the actor. Finally, handling channels invisible to the system, viz., shared memory, is a challenge in itself. We address these issues in Sections 4 and 6.

We have implemented Causeway in the FreeBSD operating system kernel, the `libpthread` and the `libevent` [8] libraries. Causeway, thus, achieves automatic propagation of metadata without the need for application modification.

Using Causeway we could rapidly implement a distributed priority scheduling system where priority of a task is injected and propagated as metadata, and accessed to implement global priority scheduling. This required writing only about 150 lines of code on top of Causeway. With this system we obtained global priority scheduling on an implementation of the TPC-W [10] benchmark used as a test distributed program. This distributed program includes a Web server, an application server and a database, all running on different machines. Each request for service is assigned a priority. This priority is then passed as metadata which follows *all* actors performing the execution for this request in the Web server, application server and the database. No modification of the TPC-W benchmark, other than selective injection of priority, was required.

Causeway is not the first system to advocate the propagation of metadata along request execution paths in distributed systems. Earlier work in Domain and Type Enforcement (DTE) in Unix systems [2] and Stateful Distributed Interposition (SDI) [9] employ metadata or context propagating mechanisms similar to Causeway. While DTE propagates the type of data written by a sending process and the domain of the sending process for interprocess communication to implement security policies, Causeway extends this mechanism to propagate arbitrary types of metadata across different kinds of channels for a variety of meta-applications. The work closest to Causeway is SDI [9] which also provides metadata or context propagating mechanism for multitiered servers. Causeway differs from SDI in two aspects: first, Causeway propagates the value of the metadata across channels and not its reference as in SDI, and, second, we want to extend Causeway to handle shared memory channels. Shared memory channels occur frequently in many programs, e.g., Apache and MySQL which are used extensively to build distributed applications.

There have been customized solutions to build some meta-applications as follows. Aguilera et al. [1] infer causal paths from message traces to locate nodes causing performance bottlenecks. The use of request tagging has been utilized to determine faults in Internet services [4]. The resulting Pinpoint system uses instrumentation of the J2EE platform to pass on request identifiers among the different components of the system. These meta-applications, and many more, can be implemented on top of Causeway.

Magpie [3, 5] represents a different approach to the analysis of distributed programs. Magpie logs events, and extracts events belonging to a particular request execution by performing temporal joins over this log. These joins are based on application-specific schemas, which may require considerable expertise and knowledge about the application. Magpie and request identification using Causeway present an interesting set of tradeoffs. Magpie does not require kernel or library modifications, and leverages event logging facilities already present in Windows. In contrast, Causeway accepts the premise of such modifications, and as a result avoids the need for detailed knowledge about the application.

With Causeway users can implement tasks like priority scheduling and performance debugging of distributed programs. Such users are different from the class of operating systems developers and application developers. Meta-application developers use the interfaces exported by Causeway to implement the desired meta-application requiring little knowledge of the application or the operating system. By separating development of meta-applications from applications, Causeway parallels the concept of Aspect-Oriented Programming [7] which allows developers to dynamically modify static application to achieve secondary goals without modifying the original static model.

The rest of this paper is organized as follows. We justify the need for a framework like Causeway in Section 2. We give a detailed specification of metadata in Section 3. Section 4 presents a design overview for Causeway. We give demonstration of Causeway's use in Section 5. Ongoing and future work is outlined in Section 6. We conclude in Section 7.

2 Need for a Framework

In this section we motivate why the operating system should support metadata injection, access, and propagation. In other words, we answer the question — "why not build the support into applications".

First, we note that the use of metadata is significantly different than the (application) data. Hence, from a software engineering viewpoint, there is a logical separation between how data and metadata are handled.

Second, propagating metadata at application level only will involve augmenting applications and application-level inter-process communication protocols. This approach has its own pitfalls. Consider a multi-tiered server for web services. Let us assume, an application-specific HTTP header is defined to propagate metadata to a web server. But not all applications use the same protocol. For instance, the web server may need to communicate to a database server. In this case, the database server does not understand HTTP. To propagate metadata to the database server, then, the communication protocol between the web server and the database server needs to be augmented as well. In essence, by this approach all possible application-level communication protocols will require augmentation — a tedious solution. By making the propagation of metadata a system-level function, it becomes independent of the application-level communication protocol being used.

Finally, in a distributed program, it is possible that some individual components are unaware about the presence of metadata or ignore it. Consider a 3-tier system, where the middle tier application is unaware of metadata. The front and the back-end tiers may still, however, need to access metadata. In this scenario, operating system support for automatic metadata propagation is required in the middle tier even though the middle tier application may remain ignorant to metadata.

3 Metadata

Metadata in Causeway is a five-tuple of *(identifier, type, value, propagation bit mask, merge routine identifier)*. On injection, a metadata object is created and assigned an immutable, system-wide unique identifier. Type and value are self-explanatory. Meta-applications can define new metadata types, if required. The propagation bit mask contains a flag per channel type signifying whether this metadata object is propagated across channels of that type or not. The merge routine identifier specifies which *merge routine* should be invoked, when required. A merge routine takes two or more metadata objects as input and combines them to produce a single metadata object. Causeway implements frequently used merge routines like `min, max, concat,` etc. Other merge routines can be implemented in Causeway, if required. A merge routine is invoked on the incoming metadata and the existing metadata of an actor when they have the same identifier but differ in value.

4 Causeway Design

Causeway has two components: (1) interfaces to inject and access metadata and (2) mechanisms to automate propagation of metadata.

4.1 Interfaces

Meta-applications can interact with Causeway in two ways — through an interface by which actors can inject and access metadata and through a callback interface under which Causeway calls handlers registered by the meta-application.

Actor Interface Causeway provides interfaces for injection, inspection, modification and removal of metadata by actors. These interfaces may be called from user-level or kernel-level by an actor, which could be a process, a thread or an event-handler.

Causeway defines the following interface functions to be called by an actor: `cwa_type_query` retrieves the collection of metadata types that are associated with the actor; `cwa_data_lookup` retrieves any metadata of the given type that is associated with the actor; `cwa_data_insert` associates the given metadata with the actor, overwriting any prior metadata of that type; and `cwa_data_remove` disassociates any metadata of the given type from the actor. Since all metadata are actor-private synchronization of metadata access interfaces is not required.

Callback Interface Using Causeway's callback interface the meta-application can register a *transfer-point* callback method. A transfer point is a point where data is read from or written to a channel by an actor. At a transfer point Causeway determines if the type of the metadata being passed has a callback method registered. If a callback method exists, it is invoked with the metadata as argument. The callback method reads and possibly modifies the metadata and passes it back to the transfer point. The callback method can call arbitrary operating system code, e.g., to change the priorities of actors.

4.2 Automatic Propagation of Metadata

When an actor performs a write on a channel, the actor's metadata is associated with the data written into the channel. On a subsequent read on the channel by an actor, metadata is propagated from the data and assigned to the actor. First, we describe the rules of metadata assignment to an actor. Then we describe the propagation mechanism across each of the channel types.

4.2.1 Assigning Metadata to an Actor

There are two ways metadata can be assigned to an actor - injection and propagation across a channel. On injection, an actor loses any existing metadata and the injected metadata is assigned to it. On propagation, two cases are possible. First, the actor does not have any existing metadata, or the identifier of its existing metadata does not match the identifier of the metadata propagated. In this case the actor loses its existing metadata, if any, and the propagated metadata is assigned to it. Second, the identifier of the actor's existing metadata matches that of the propagated one but the metadata values are different (no action is required if the values match). In this case the merge routine, specified in the metadata, is invoked on the two metadata, and the result is assigned to the actor.

4.2.2 Propagation across Channels

Now we describe the propagation mechanism across each of the channel types. We emphasize that the rules described in Section 4.2.1 are applied to assign metadata to an actor after propagation across a channel. Causeway currently implements metadata propagation across sockets and pipes.

Sockets and Pipes Causeway handles sockets and pipes similarly. When an actor writes to a socket (or a pipe), Causeway associates metadata from the actor to the data written. On subsequent read from the socket by another (or the same) actor, metadata is propagated from the data to the actor.

The above applies for `LOCAL` sockets only. For `INTERNET` sockets, data is encapsulated in IP packets for send and receive across sockets. Causeway encapsulates metadata, in addition to data, in the IP packets. For IPv4, Causeway encapsulates metadata in the IP header as IP options. In particular, Causeway defines a new IP option type, populates the IP header with the option type, option length, and option payload. At the receiving side, the metadata, if any, is extracted from the IP options. Since IP options can be a maximum of 40 bytes only, with 1 byte each for options type and options length, Causeway can transfer at most 38 bytes of metadata via this mechanism. For most practical purposes, this has proven sufficient. This limitation is an artifact of Causeway's implementation and not its design. A general purpose tunneling protocol could be used to overcome this limitation, if required. For IPv6, Causeway uses the destination options in the IP header which does not have any size limitation. Further details about that are outside the scope of this paper.

The following case presents a challenge to the above design. Consider a scenario where multiple pieces of data are ready to be read from a socket (or pipe), and at least one piece has a metadata identifier different than rest of the above. Then a decision needs to be made about what metadata is to be propagated to the actor reading from the socket (or pipe). Causeway resolves this situation as follows. The pieces of data ready on the socket are read in a `FIFO` manner. Causeway returns from the read just before the first piece having metadata identifier different than the earlier pieces. So, all the pieces of data read by the actor are guaranteed to have the same metadata identifier. The merge routine is then applied on these metadata, if their values differ, and the result is propagated to the actor. In our implementation of Causeway on FreeBSD, we associate metadata with `mbuf`s on send and receive operations on `socket`s.

5 Using Causeway

Meta-applications to control and analyze the execution of distributed programs can be built easily using Causeway. We illustrate two such meta-applications here: a multi-tier priority scheduling system and a distributed profiler.

5.1 Multi-tier Priority Scheduling System

Using Causeway we could rapidly implement a multi-tier priority scheduling system, controlling the order in which requests sent to a multi-tiered, web-based application server are executed. Under this system, the application injects priority as metadata, Causeway automatically propagates the priority metadata to all the tiers, and the meta-application uses the priority metadata to enforce priority scheduling on each tier. The meta-application is automatically invoked on each tier through Causeway's callback mechanism.

The implementation of this system on top of Causeway required writing only about 150 lines of code. We tested this system with an implementation of the TPC-W benchmark [10]. No modifications were made to the TPC-W code, other than selective injection of priority. We subjected the TPC-W system to a background workload and a foreground test load. The background workload was injected with metadata signifying default priority. The foreground load was injected with metadata for default priority in one case, and high priority for another. Response time measurement for the foreground load showed one to two orders of magnitude of improvement when using high priority.

5.2 Distributed Profiler

In this section we present the design for a distributed profiler that we are developing using Causeway. A distributed application has multiple components executing in different processes. Furthermore, these different processes may be executing on multiple machines. While it is possible to profile the components in isolation, it is hard to collate the profile information for different components to form a single, global profile. We intend to achieve this with a distributed profiler: we will pass context information as metadata on remote procedure calls (RPC) between the application components using Causeway, and then using this context information we will stitch together the profile information for the components to a generate a single, global profile.

6 Future Work

In this section we describe the design of Causeway to propagate metadata across file and shared memory channels. As ongoing work, this design is being implemented in Causeway. As future work, we intend to extend the design of Causeway to handle parallel computation paths and address security concerns. Finally, we wish to quantify the overhead of using Causeway.

6.1 Files

When an actor writes to a file, Causeway assigns the metadata from the actor to the range of bytes written. On a read operation, two cases are possible: (1) All the bytes read are associated with the same metadata - The metadata is propagated to the actor in this case, (2) At least one byte has associated metadata different than the rest - In this case the merge routine, specified in the metadata, is applied on the different metadata, and the result is propagated to the actor.

6.2 Shared Memory

Producer-consumer is a popular model of shared memory usage. This model is used, by applications like Apache and MySQL. At an abstract level, the model works as follows. Producers and consumers share a buffer or queue of objects. A producer creates an object, acquires a lock to enter the critical section, adds the object to the shared buffer or queue, and releases the lock. A consumer acquires a lock to enter the critical section, retrieves and removes an object from the shared buffer or queue, releases the lock, and then accesses the retrieved object. The use of system-supported synchronization primitives, like `pthread_mutex` or `pthread_rwlock`, can make producer-consumer communication through shared memory visible to Causeway.

We note that the producer accesses the created object just *before* the *lock* operation and in the critical section, while the consumer accesses the retrieved object in the critical section and just *after* the *unlock* operation. We are investigating ways to identify this pattern and insert (in the source or precompiled binary) calls to save metadata from the producer and calls to retrieve metadata in the consumer. The transformed producer code will do the following: create the object, save the producer's metadata and associate it with the created object; then enter the critical section as in the unmodified program. After the critical section, the transformed consumer code will do the following: access the retrieved object and retrieve the metadata associated with the retrieved object.

6.3 Execution Path Fork and Join

Causeway needs to handle execution path *fork*s and *join*s caused by parallel computation paths. In the common case, an actor writes to a channel and then reads from the same channel, waiting for a response. However, sometimes, an actor may write to multiple channels without waiting for the individual responses. As an example, a web server may send queries to multiple nodes in a replicated database system and then wait for their individual responses. Each of these writes constitutes a fork in the execution path. When the response corresponding to a fork arrives, it is termed a join. In the above example, the response from a database server constitutes a join. As future work, we intend to extend the design of Causeway to identify and handle such forks and joins in the execution paths.

6.4 Security Concerns

Like SDI [9] we argue that the issue of illegal network access modifying metadata in IP packets should be addressed by using IPSec [6]. In order to prevent the illegal modification of the metadata by the application, we intend to incorporate a secure signing mechanism like MD5 as a part of the metadata for propagation across the user-kernel boundary.

7 Conclusions

The contributions of this paper are the following. We have designed Causeway, operating system support for facilitating development of meta-applications, like priority scheduling and performance debugging, to control and analyze the execution of distributed programs. Causeway provides interfaces for metadata injection and access and performs automatic propagation of metadata in distributed programs. Propagated metadata can be accessed and used to implement the desired service in the system. We have implemented Causeway in the FreeBSD operating system, the `libpthread` and the `libevent` libraries. We have demonstrated the use of Causeway by implementing a multi-tier priority scheduling system and using it to achieve global priority scheduling on an implementation of the TPC-W benchmark [10].

References

[1] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance Debugging for Distributed Systems of Black Boxes. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, pages 74–89, Oct. 2003.

[2] L. Badger, D. F. Sterne, D. L. Sherman, K. M. Walker, and S. A. Haghighat. A Domain and Type Enforcement UNIX Prototype. In *Fifth USENIX UNIX Security Symposium*, June 1995.

[3] P. T. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using Magpie for Request Extraction and Workload Modelling. In *OSDI*, pages 259–272, Dec. 2004.

[4] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: Problem Determination in Large, Dynamic Internet Services. In *Proceedings of the International Conference on Dependable Systems and Networks (IPDS Track)*, pages 595–604, June 2002.

[5] R. Isaacs, P. Barham, J. Bulpin, R. Mortier, and D. Narayanan. Request extraction in Magpie: events, schemas and temporal joins. In *SIGOPS EW'04: ACM SIGOPS European Workshop*, Sept. 2004.

[6] S. Kent and R. Atkinson. Security Architecture for the Internet Protocol. In *IETF RFC 2401*, 1998.

[7] ONJava.com. Introduction to Aspect-Oriented Programming. At http://www.onjava.com/pub/a/onjava/2004/01/14/aop.html.

[8] N. Provos. Libevent - an event notification library. Version 0.7c is available from the author's web site, http://www.monkey.org/∼provos/libevent/, Oct. 2003. Libevent is also included in recent releases of the NetBSD and OpenBSD operating systems.

[9] J. Reumann and K. G. Shin. Stateful Distributed Interposition. *ACM Transactions on Computer Systems*, 22(1):1–48, Feb. 2004.

[10] T. P. P. C. (TPC). TPC BENCHMARK W (web commerce). At http://www.tpc.org/tpcw/, Feb. 2002.