State Management in .NET Web Services

Xiang Song, Namgeun Jeong, Phillip W. Hutto, Umakishore Ramachandran, James M. Rehg
Center for Experimental Research in Computing Systems
College of Computing, Georgia Institute of Technology
Atlanta, Georgia 30332
{songx,namgeun,pwh,rama,rehg}@cc.gatech.edu

Abstract

In the paper, we identify a problem for certain applications wishing to use the web service paradigm to enhance interoperability: rapid, robust state maintenance. We classify two kinds of state: application state and session state. While many features are available to support session data, special mechanisms for application state maintenance are less well developed. Application state maintenance is integral to providing reliable, fault-tolerant web services. We discuss three different models to solve the problem and compare the advantages and disadvantages of each. Experimental results show that the choice of which model to use depends on application requirements. Many important emerging applications will involve the communication of potentially large time-sequenced data streams among heterogeneous clients with varying QoS requirements. D-Stampede.NET is an implementation of a system designed to support the development of such applications. We describe our web service implementation along with our state server solution to the application state management problem. A simple demo application is described and measured to validate performance.

1. Introduction

Peer to peer computing (P2P) is becoming an increasingly prominent aspect of ubiquitous computing environments. P2P file transfer systems (such as Freenet or Gnutella), sensor networks and other pervasive computing systems are popular and useful but support limited interoperability. Peers in disparate systems often are unaware of services available in another. One way to enhance the interoperability of these systems is to by providing web service interfaces. Web services also support a well-defined service discovery infrastructure.

Web services are “software systems designed to support interoperable machine-to-machine interaction over a network” [1] typically using SOAP-encapsulated messaging, HTTP transport and XML serialization. Web services promise the exact features needed for P2P interoperability. They support WSDL and UDDI for remote service discovery and XML for interoperability.

However, one problem of web services is the difficulty of state maintenance. The web services paradigm is a classic example of stateless client-server interaction and is built on top of the stateless HTTP protocol. By stateless, we mean that client requests are independent and no memory of previous client requests is required for servicing new requests. This design mirrors that of file servers like NFS [16] and simplifies recovery after failure. (No recovery dialogue is required with former clients after a crash.) Note that state can be maintained externally by the web service but is not required by the HTTP protocol stack. However modern software like P2P applications frequently need state maintenance, and state must be maintained on both sides of the communication since communicating entities are peers. Therefore, it is important for many types of applications such as pervasive computing applications, that a flexible and efficient state maintenance

capability be provided.

There are generally two kinds of state that might need to be saved on the server: session state and application state. Recent requests from a particular client process are often considered in the same session. Session state data is only visible within that particular session. The familiar shopping cart on commercial websites provides a good example of session state. Users select several products that they want to buy into a shopping cart and when they click "Add to Cart" button, the product is added into the session state. Adding more items does not remove previous items and session state is only visible to the individual shopper. Session state is retained by the service for an extended period until some reasonable expiration time has passed. Session state is per-user while application state is per-application, Note that web servers often support multiple applications simultaneously. Application state is shared across sessions within the same application but is not shared between distinct applications or application instances. Application state is typically maintained in an external database. Common examples include product information (descriptions, prices, images, etc.) for services like Amazon.com and eBay.

Session state maintenance is the subject of lively debate in the web programming world and a variety of techniques (cookies, state servers, etc.) are used to implement session state. But application state maintenance has been less well explored. The standard solution is to store most application state in a database and a variety of techniques exist for facilitating the interaction of web servers and databases. Sophisticated optimizations such as data caching are also employed. But external databases are not well-suited for storing application state for certain types of increasingly important applications. Pervasive computing and sensor-based environments often involve dynamic, P2P interactions. Many such systems can be viewed as managing streams of potentially high-volume data. For example, surveillance applications require coordination and processing of a large number of sensor streams (video, audio, motion data, etc.). These applications often involve platforms with varying computational abilities (embedded systems, handhelds, servers, etc.) and sometimes require computationally intense processing (cluster-based image recognition, etc.). In short, such applications require:

- High-bandwidth interactions (relatively large amounts of data must be frequently transferred),
- Low-latency interactions (data delivery must be timely).

An additional interesting characteristic of these environments is variable QoS. Much data is uninteresting and redundant. The occasional loss of an audio or video frame is not likely to be a problem but the overall quality of the computation must not be degraded by too much data loss.

Flexible application state management is important because of the need for:

(1) Performance: Database storage may be too slow to satisfy throughput and latency requirements of the emerging class of applications we consider.

(2) Fault-tolerance: Several web services are frequently implemented using a single web server resource. Web services are sometimes ill-behaved, requiring occasional maintenance and restart of web servers. Memory-based application state will be lost by these administrative restarts. In general, web services should provide some form of fault-tolerance. (As an example .NET web services use a mechanism, named process recycling, to solve the problem above. .NET may tear down the web services worker process from time to time to make it "healthy".)

(3) Persistence: While much data in pervasive computing environments is ephemeral, it is important to persist data periodically for subsequent analysis and processing.

Solving the state management problem with fault tolerance on web services is a complex problem. In this paper, we propose our first step toward this goal. We begin to explore the problem by examining three different solutions to the application state maintenance problem and compare their performance. We also discuss appropriate applications of each model based on the experimental results. As an example, we describe which application state management solution is most appropriate for supporting D-Stampede [5], a parallel/distributed computing middleware infrastructure for supporting pervasive computing applications developed at Georgia Tech.

The rest of the paper is organized as follows: first, we introduce the three models that might be used to solve the problem. Second, we discuss some preliminary performance measurements. Third, we show how we have integrated application state management into a web services implementation of D-Stampede on the .NET platform. Finally, we show the related work and draw some conclusions.

2. State Management Model in Web Services

In a typical implementation of web services (such as the .NET implementation), all web applications execute in a single worker process space (Figure 1). A single application error may crash the process, leading to the failure of all co-resident applications.

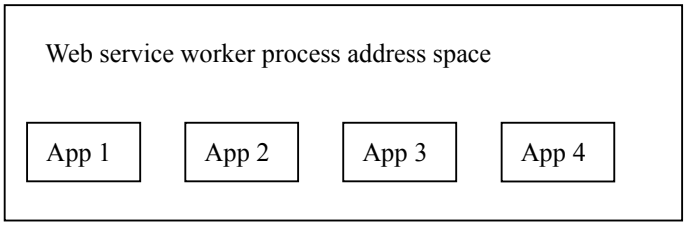

Figure 1: Co-resident Web Service Applications

We consider three possible solutions to the state maintenance problem for web services. We call these the *state server* model, the *database* model, and the *proxy* model. Each mechanism maintains state outside of the web service worker process. Details of each model and some qualitative and quantitative comparisons follow.

2.1 State Server Model

To avoid application failure coupling, it would be possible to simply run a single application in each worker process address space. However, this solution still does not handle administrative shutdown and restart of the web service gracefully. We can solve both problems by maintaining server state in another process on the same machine (Figure 2). We call the second process a State Server. Application state is check-pointed and maintained in the separate State Server. Application state can be retrieved from the State Server as necessary when the web service restarts.

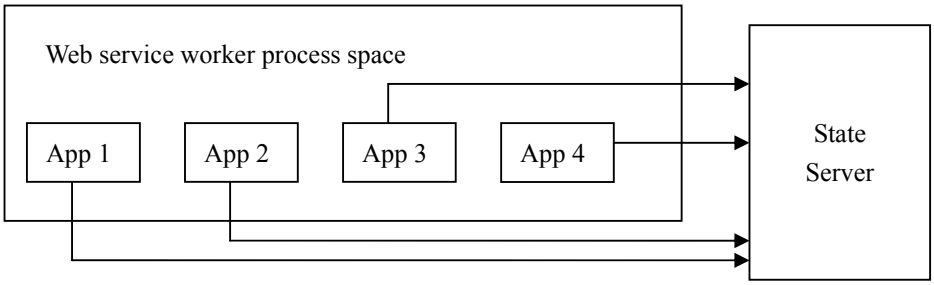

Figure 2: State Server Model

2.2 Database Model

Another common solution is to store application state in a database. State access is now performed indirectly through database queries and updates. State maintenance issues are effectively delegated to the database but with an increase in complexity and a potential decrease in performance. The database model naturally supports persistence of application state.

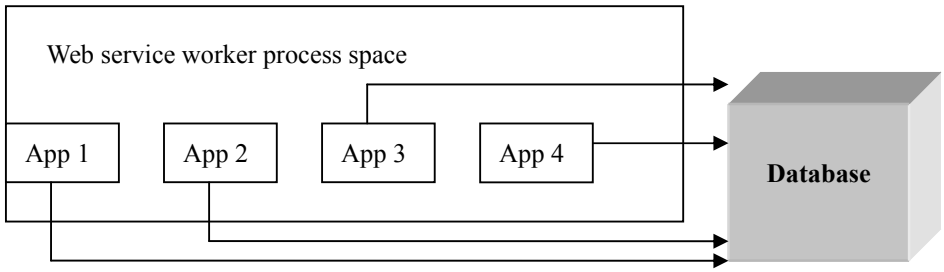

Figure 3: Database Model

2.3 Proxy Model

In the first approach, application state was moved from inside the process address space to an external process. In the second model, application data is moved into an external (persistent) data store. In the third approach application logic as well as state is migrated outside the web service infrastructure. In this model, the web service worker process simply proxies application requests to a third-tier computation. The proxy is freed from almost all state maintenance concerns. The web service itself acts, in this case, merely to provide interoperability. Some computation in the form of data and protocol conversion can be done within the proxy itself as necessary.

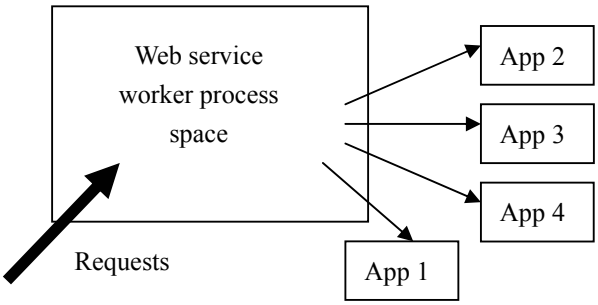

Figure 4: Proxy Model

2.4 Comparison of the Models

Compared to in-process state management, all three models have some additional overhead. But all of them provide robustness to the crash of the web services worker process. The probability of failure increases for complex applications that require frequent large data transfers and share the same worker process space. All three models provide advantage for such scenarios.

All three models can potentially be implemented using either primary or secondary storage. However, the database model is most naturally implemented using disk and the other two models are most naturally implemented using memory. Disk-based storage is obviously persistent and has the added benefit of surviving machine restart. However, it is likely that database access may be slower than the other two mechanisms. Additional processing time may be required, as well, to convert data into a form suitable for storage in the database.

The proxy model is simpler and easier to implement than the state server model. Applications can be easily made visible using a web services proxy interface with a relatively small amount of effort. The proxy, however, must still handle SOAP and XML processing as well as any necessary protocol conversion. Another problem with the proxy model is that future web service enhancements such as security, coordination, reliable messaging will not be available to the application since the application is actually hosted outside the web services environment.

The state server model falls in the middle ground between the proxy and database models. The application still resides in the web services environment but the state is stored outside and in-memory, for robustness and performance. Additional isolation can be achieved if the state server resides on another host (although this increases the chance of partial failure). Another advantage of this model over the proxy model is that processing data before storing it may save space and time. Consider a producer/consumer scenario involving the transfer of bitmap (BMP) images, in which intermediate data is stored in a web service, as shown in Figure 5. If the web service compresses images using JPG before saving them to the state server, significant savings are achieved.

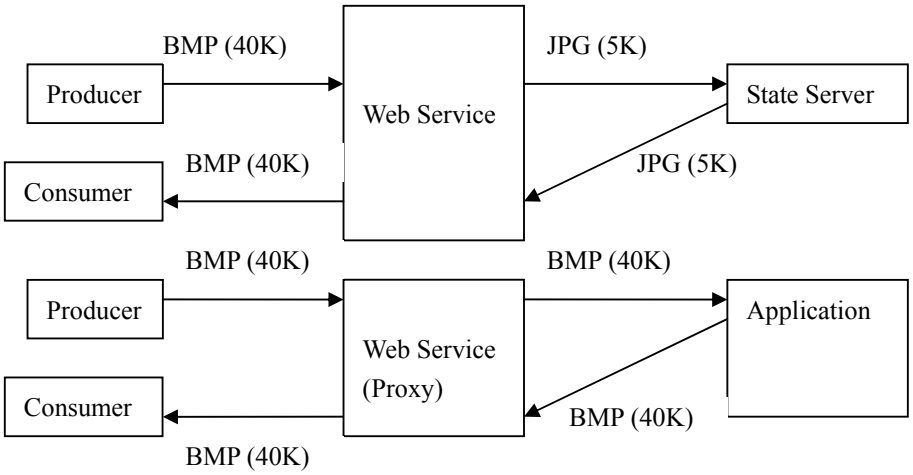

Figure 5: Advantage of State Server Model over Proxy Model

Each image transfer reduces transfer time and space required for saving the image in the state server.

The efficiency of the state server model clearly depends on the granularity and frequency of state transfers. In the extreme this model is infeasible. Reasonable state access patterns must be developed on a per-application basis. A disadvantage of the state server model is its sensitivity to state transfer access patterns and the additional complexity it requires for tuning transfer frequency.

2.5 Micro Measurements of the State Management Models

In order to understand these models better, we present the results of a simple set of experiments.

2.5.1 Experimental Setup

We use two Pentium 4 computers: 1.8GHz CPU, 1GB physical memory and 100 Mbps Fast Ethernet network interface. We have Microsoft Windows XP Professional Edition (machine 1) on one machine and Microsoft Windows Server 2003 Enterprise Edition (machine 2) on the other. To provide Web services, Microsoft Visual Studio .NET 2003 is installed on each machine. To support the database model, Microsoft SQL server 2000 runs on machine 2. For other models, the state server process for the state server model and application process for the proxy model run on machine 2 as well, and .NET Remoting facilitates the communications between the web service and the back-end processes.

2.5.2 Sample Program

We build a simple producer-consumer style client application and web services to obtain the basic performance characteristics of the three models. The client (running on machine 2) *put*s indexed data items through the web service to the server and the server stores all of them using each of the three methods mentioned above. Except for the proxy model, each data item is compressed and stored. The compression ratio is set to eight. The client then *get*s the data item with a specific index from the server. In the DB server model, the web service sends an SQL query to the SQL server to retrieve the data item using the index.

2.5.3 Results and Analysis

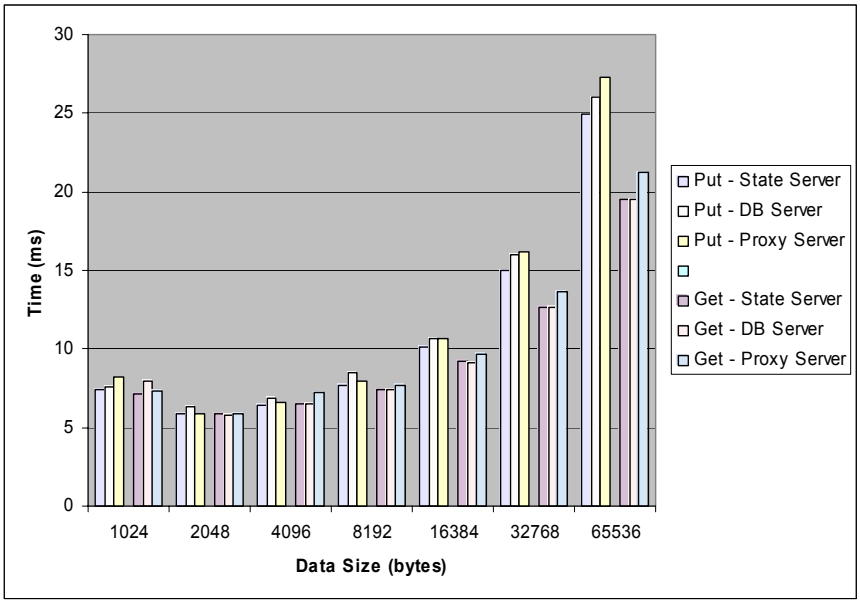

Figure 6 Comparison of Put/Get Operation Performance

Figures 6 show the average response time for 1000 iterations of put and get operations measured by the client application for each model. The results show that the state server model performs better than the other two models and the proxy model performs the worst in most cases. For put operations, the state server model is 2~10% faster than the database model. Considering that the SQL Server has to write each transaction log to the disk, the database model performs very well and seems to be quite scalable. For get operations, these models show about the same performance. The database model appears to be more efficient than we had initially expected. We surmise that I/O scheduling is highly optimized within SQL Server and that the IPC mechanism used in the state server implementation (.NET Remoting) is less efficient than expected.

Note that our experiments are by no means attempts to simulate any realistic web-service usage patterns. Since it was performed to understand the basic differences among the three models, many other aspects have to be investigated further. When the SQL server and back end processes are running on an SMP machine, and there are multiple concurrent web service requests, each model's performance has to be measured in terms of throughput as well as latency. The overhead of flushing data to the disk should also be evaluated with the characteristics of the file system.

3. D-Stampede.NET

To validate the feasibility of the state server approach, we have added a simple state server to a pervasive computing programming environment, called D-Stampede, developed at Georgia tech and currently being used for a variety of applications. The D-Stampede.NET project is a re-implementation of the D-Stampede distributed/parallel programming infrastructure on the .NET platform using web services. In this section, we describe D-Stampede and discuss the advantages of implementing it using .NET web services. We also present some problems encountered during design and coding and our solutions. We focus on the state maintenance components of D-Stampede.NET. A full report on the implementation will be provided in a later paper.

3.1 What is D-Stampede?

Stampede is a distributed programming system that supports communication of time-stamped items between spatially distributed nodes [5]. An API is provided to simplify the programming of time-sensitive applications and to hide low level communication details. Stampede provides global communication abstractions called channels, queues, and registers that are visible to all threads in cooperating processes spread across heterogeneous hosts. Participating threads get and put items of arbitrary types. Data items are timestamped and automatically reclaimed by a global garbage collection mechanism. Channels are commonly used for sending streams of audio or video data and other sensor-based environmental readings (temperature, light, etc.). Some applications produce large volumes of data and have low latency requirements. D-Stampede extends the original cluster-based Stampede to allow heterogeneous clients to dynamically join and leave ongoing applications to better support sensor networks and pervasive computing environments.

We chose to re-implement D-Stampede as a .NET web service for several reasons:
(a) interoperability with existing web services and applications (web service interfaces are increasing in popularity among businesses and within specialized areas like the GRID computing community),
(b) built-in service publication and discovery (D-Stampede does not have a general service announcement and discovery capability),
(c) ease of development (the .NET platform and Visual Studio IDE have many features to accelerate application development),
(d) interoperability with other services (using web services as a common language),
(e) evaluate suitability of web services for pervasive computing (determine if existing web service frameworks are sufficiently flexible and efficient to support pervasive computing communication requirements).

3.2 D-Stampede.NET Architecture

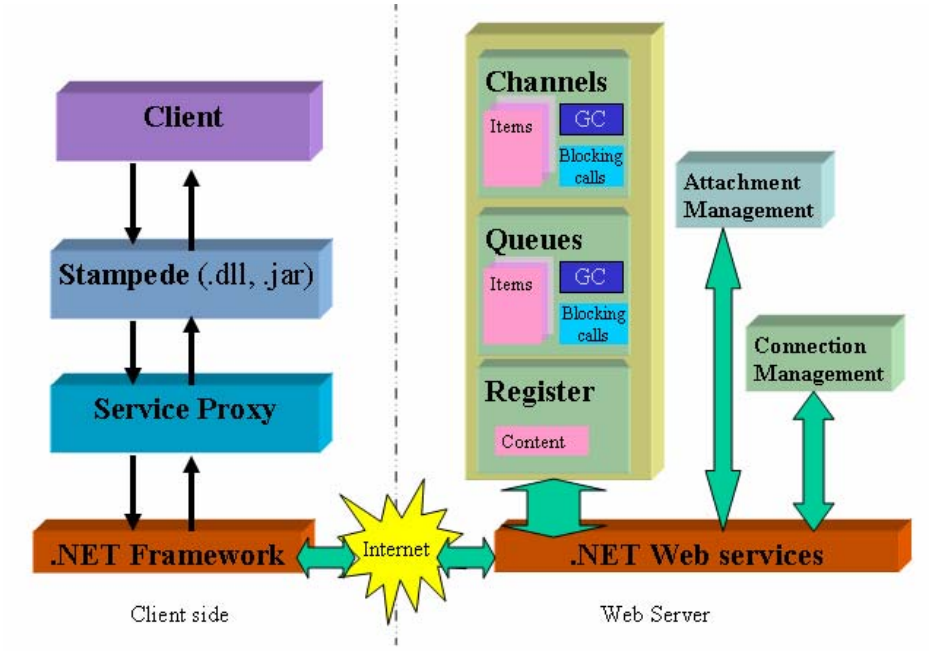

Figure 7: D-Stampede.NET Architecture

The D-Stampede.NET server (web server) is implemented on top of the .NET platform. D-Stampede APIs are provided by a client-side D-Stampede library (.dll or .jar). This library calls out into an automatically generated web services proxy. This client-side proxy initiates communication with the web service using lower level .NET communication primitives. These lower level services encode D-Stampede data using XML and encapsulate the XML in a SOAP envelope for delivery (via HTTP) to the web service. On receipt, the web service unpacks the request and performs the requested action (e.g. channel get or put).

D-Stampede supports blocking I/O and we have carefully coded the web service to efficiently and scalably support a large number of blocking requests. Recall, also, that D-Stampede supports transparent garbage collection to reclaim inaccessible items in channels, queues and registers. D-Stampede supports a notion of "attaching" to a channel. We have also introduced a connection abstraction in the .NET implementation. Attachment management and connection management are used to record the connection between clients and server or attachment between clients and data containers (channels, queues or registers). These are examples of server state currently maintained using application state management.

3.3 State Management Problem and Solution

As mentioned earlier, a significant amount of state needs to be maintained in the D-Stampede.NET web service, such as channel data, and connection and attachment information. The .NET framework (ASP.NET specifically) provides a mechanism for saving session state outside the worker process, which is suitable to keep the per-client information such as attachment and connection. Application state, on the other hand, is only stored in-process and can be lost due to a restart. In addition, .NET provides another mechanism called process recycling that periodically tears down the worker process in order to recover from application failure and memory leaks. All state information is lost when this happens. So some form of enhanced state management is required for D-Stampede.NET.

We chose the state server model for D-Stampede.NET for the following reasons:

(1) As we have seen in the previous section, preliminary results show that the state server model yields some performance advantage for *put* operation over the other two models; applications that are built on top of Stampede tend to produce (*put*) more than they consume (*get*),
(2) The state server could be placed on a separate machine, providing enhanced fault isolation,
(3) The state server model works well in SMP systems (Stampede was originally designed to run on clusters and contains a variety of cluster-related optimizations. Stampede processes running on distinct processors in an SMP system all have fast access to state saved in the shared memory of the system).

In the following section, we show and discuss some preliminary performance measurements of our D-Stampede.NET implementation using a state server.

3.4 Performance

3.4.1 Experimental Setup

To test the performance of D-Stampede.NET, we built a demo application that includes two video producers, an image processor and a consumer (Figure 8). The producers capture images from a web cam and put each frame as a data item into a particular channel. The image processor gets the latest images from both channels, mixes them together using a couple of different compositing algorithms and then puts the processed frame into another channel. The consumer simply retrieves and displays processed frames. This demo can be

considered a microcosm of a teleconferencing or a video surveillance application.

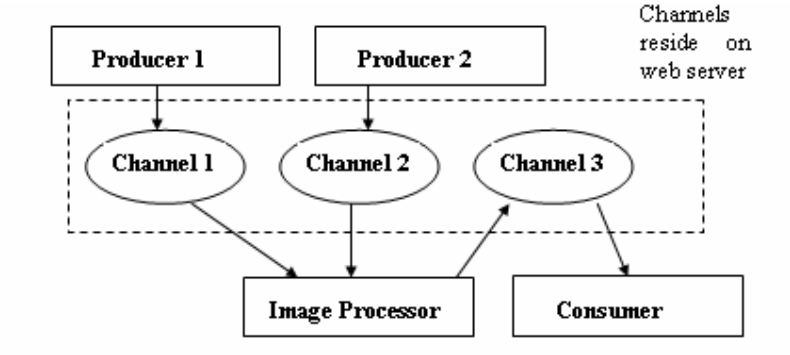

Figure 8: D-Stampede Demo Setup

To measure the performance, we use the same hardware environment as in the micro measurements. One producer and the image process run on the Windows 2003 machine and all others, including the web server (and web service), run on the Windows XP machine.

Basic tests with this setup yields a throughput of 12-14 frames per second at the consumer, which is quite a good for the kinds of applications that are envisioned with the Stampede framework. The latency for displaying an image from the time of capture at the producer is around 15 ms. These are preliminary results and more detailed experiments are underway. Suffice it to say, that these latency and throughput results attest to the viability of using the Web Services paradigm for such demanding applications.

4. Related Work

State management has been a topic of discussion in computer science for quite some time. The concept of stateful and stateless systems goes back all the way to the earliest works in automata theory. The state of an automaton (state machine) can be regarded as encoding the entire input history of that machine and internal state can be greatly reduced if the machine has access to the input history.

Thus, state can be viewed as the “memory” a system maintains of recent interactions. A stateless system has no such memory by definition and is unable to participate in extended “dialogues” or conversations where subsequent outputs depend on earlier inputs. Note that “stateless” is a bit of a misnomer and that such systems may still access a variety of stored items. For example, a “stateless” file server can still send and receive files and a “stateless” web server can still send and receive web pages. These data items can certainly be viewed as state of some form but the term stateless typically refers specifically to memory of previous interactions with clients, that is, “conversational state.”

The topic of state management has periodically resurfaced through the years, most notably in discussions of the design of stateless servers (NFS), stateless protocols (HTTP), stateless objects, session state (shopping carts), stateless messaging (UDP), stateless QoS (Differentiated Services) and stateful packet filters. Indeed, most applications are inherently stateful but stateless implementations and interactions provide important benefits in some contexts.

For example, the NFS server [16] was designed to be stateless to eliminate the need for complex recovery logic. If no persistent record of ongoing client conversations is recorded, then there is no work to do after a crash to re-activate or abort those conversations. Additional benefits are provided as well: not having to

maintain conversation information reduces the resource requirements of the server and simplifies server logic; both of these benefits allow servers to scale and handle larger numbers of concurrent clients. Stateless servers are particularly good at handling very high numbers of concurrent requests. Certain disadvantages also result: recovery logic is pushed out of the server and into clients who must continue to retry failed operations; operation duplication is possible for non-idempotent operations; individual requests typically require more state (parameters) and take longer to complete since operations such as authentication cannot be performed just once at the beginning of a session. Note that stateless design was abandoned [15] in the latest version (NSF-4) of the protocol. AFS is an example of a stateful distributed file server that uses server state to support disconnected operation.

Stateless servers interact with clients via stateless protocols. Most common internet protocols such as FTP and NNTP are stateful while the NFS and HTTP protocols are stateless.[17] Note that it is possible for a server to require clients to maintain and manage their own session state. The well-known cookie mechanism [7] used by web servers employs this technique.

Statefulness is closely related to persistence and fault-tolerance. Stateful servers must checkpoint or persist their state data periodically to allow recovery and forward progress. Transactions and reliable messaging are both inherently stateful. Proposed web service extensions in both areas (WS-TRANS [4], WS-ReliableMessaging [13], WS-Reliability [14]) require servers to maintain some state.

Persistence has its own benefits and motivations, in addition to maintaining server state across failures. Persisting data allows for logging and later analysis and retrieval. A variety of frameworks such as Java session and entity beans [18] have been proposed for the management of persistent data in data-intensive web-based applications. The ADO [10] and JDO [9] “data object” frameworks provide an object-oriented, memory-based abstraction of persistent data. Various persistence “providers” (such as file stores, relational or object-oriented databases) can be plugged in to actually manage persistence. In some cases, persistence management is transparent to the programmer and is introduced into the executable by post-processing generated byte codes after compilation.

Web servers (IIS, Apache) and web application execution environments (J2EE, ASP.NET) typically provide support for shopping-cart style session state management. In ASP.NET session state [11] is accessed by simply indexing into a globally visible associative array. That state may be maintained InProc (fast), or in-memory by an external StateServer (aspnet_state), or by an external database (SqlServer). In J2EE, session state is available via an HttpSession object passed to processing code. Session data is typically maintained in-process and is reclaimed after a pre-defined (arbitrary) timeout period.

Web caching is an important optimization that does not violate the statelessness of a server. Cached data serves merely to speed-up subsequent accesses but is not required for correct performance. Therefore it is not considered session state. Many web caching schemes are in use [12].

Fowler [8] has a good discussion of session state alternatives and identifies three common session state patterns: Client Session State, Server Session State, and Database Session State. He identifies two solutions to the problem of supporting session data in a web cluster (or farm in Microsoft parlance): session migration (move the session data) or server affinity (move the client processing).

5. Conclusion and Future Work

In the paper, we identified a problem for certain applications wishing to use the web service paradigm to enhance interoperability: rapid, robust state maintenance. We classify two kinds of state: application state and session state. While many features are available to support session data, special mechanisms for application state maintenance are less well developed. Application state maintenance is integral to providing reliable, fault-tolerant web services.

We discussed three different models to solve the problem and compare the advantages and disadvantages of each. Experimental results showed that the choice of which model to use depends on application requirements.

We believe that many important emerging applications will involve the communication of potentially large time-sequenced data streams among heterogeneous clients with varying QoS requirements. Such applications are common in richly sensored environments and in pervasive and context-sensitive applications such as surveillance, location tracking, habitat monitoring. D-Stampede.NET is an implementation of a system designed to support the development of such applications. We have described our web service implementation along with our state server solution to the application state management problem. A simple demo application was described and measured to validate performance.

Many issues still remain to be addressed. This work simply identifies an interesting unsolved problem and makes a few first steps towards a comprehensive solution. We need a better understanding of the various performance tradeoffs across a variety of actual workloads. We need more experience developing pervasive computing applications that utilize a web services framework with state management to better understand the code patterns that arise from such applications. This will help refine and optimize our implementations. It is quite possible that we can enhance the D-Stampede API to better support transient failures and occasional lost data. In addition, we need to develop a (simple) crash recovery protocol for D-Stampede.NET. One promising future direction involves exploring the use of in-memory databases and the replication of state servers to further enhance performance and reliability.

Longer range possibilities include incorporation of security features and forthcoming web service enhancements such as reliable messaging and transactions. In a related project we are exploring the use of GRID technologies for pervasive computing environments and hope eventually to blend our work on web services into that effort as well.

References

[1] *Web Services Architecture* - W3C Working Draft 8 August 2003, http://www.w3.org/TR/2003/WD-ws-arch-20030808/

[2] *Web Services Conversation Language (WSCL) 1.0* - W3C Note 14 March 2002 http://www.w3.org/TR/2002/NOTE-wscl10-20020314/

[3] *Web Services Coordination (WS-Coordination)*, Felipe Cabrera, George Copeland, Tom Freund, Johannes Klein, David Langworthy, David Orchard, John Shewchuk, Tony Storey. BEA, IBM and Microsoft, 2002.

http://www-106.ibm.com/developerworks/library/ws-coor/.

[4] *Web Services Transaction (WS-Transaction)*, BEA, IBM and Microsoft, 2002. http://www-106.ibm.com/developerworks/webservices/library/ws-transpec/.

[5] Sameer Adhikari, Arnab Paul, and Umakishore Ramachandran. D-Stampede: Distributed Programming System for Ubiquitous Computing. 22nd ICDCS, July 2002.

[6] Microsoft, .NET Framework Developer's Guide, http://msdn.microsoft.com/library/en-us/cpguide/html/cpconsessionstate.asp?frame=true

[7] David Kristol. *HTTP Cookies: Standards, Privacy, and Politics*. ACM Transactions on Internet Technology, Vol. 1, No. 2, November 2001, Pages 151-198.

[8] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2003.

[9] David Jordan and Craig Russel. *Java Data Objects*. O'Reilly, 2003.

[10] David Sceppa. *Microsoft ADO.NET (Core Reference)*. Microsoft Press, 2002.

[11] Alex Ferrara and Matthew Macdonald. *Programming .NET Web Services*. O'Reilly, 2002.

[12] Michael Rabinovich and Oliver Spatscheck. *Web Caching and Replication*. Addison-Wesley, 2001.

[13] Ruslan Bilorusets, et al. *Web Services Reliable Messaging Protocol (WS-ReliableMessaging)*, IBM DeveloperWorks, http://www-106.ibm.com/developerworks/webservices/library/ws-rm.

[14] *Web Services Reliability (WS-Reliability) Version 1.0*. Sun Microsystems, http://developers.sun.com/sw/platform/technologies/ws-reliability.html.

[15] B. Pawlowski, S. Shepler, C. Beame, B. Callaghan, M. Eisler, D. Noveck, D. Robinson and R. Thurlow, "*The NFS Version 4 Protocol*", Proc. 2nd Intl. System Administration and Networking (SANE) Conference, May 2000.

[16] B. Callaghan. *NFS Illustrated*. Addison-Wesley, 2000.

[17] Fielding, Gettys, Mogul, et al. *Hypertext Transfer Protocol – HTTP/1.1 RFC 2616*, Internet Society, 1999. http://www.w3.org/Protocols/rfc2616/rfc2616.html

[18] Roman, Ambler, Jewell. *Mastering Enterprise Java Beans 2e*. John Wiley & Sons, 2001.