Dean Jacobs

Data Management in Application Servers

This paper surveys data management techniques used in Application Servers. It begins with an overview of Application Server architectures and the way they have evolved from earlier transaction processing systems. It then presents a taxonomy of clustered services that differ in the way they manage data in memory and on disk. The treatment of conversational state is discussed in depth. Finally, this paper describes a persistence layer that is designed for and tightly integrated with the Application Server.

1 Introduction

Transaction processing applications maintain data representing real-world concepts and field requests against this data from client devices [1]. Transaction processing applications play an essential role in many industries, including:

- *Airlines* Managing flight schedules and passenger reservations.
- *Banking* Accessing customer accounts through tellers and ATMs.
- *Manufacturing* Order tracking, Inventory management, Planning and job scheduling
- *Telephony* Allocation of resources during call setup and teardown; Billing across multiple companies.

Typical transaction processing applications handle many short-running requests, including queries and updates. In contrast, typical non-transactional applications, such as those for scientific computing or analytical processing, handle small numbers of compute-intensive queries.

Transaction Processing Monitors provide a software infrastructure for building transaction processing applications. A key feature of TP Monitors is support for ACID properties of transactions to handle failures and other exceptional conditions. Early TP Monitors, in particular IBM CICS [2], were developed in the 1970s to run on monolithic mainframe systems. Distributed TP Monitors, such as BEA Tuxedo™ [3], were developed in the 1980s to run on collections of mid-sized computers. Distributed TP Monitors use software-level clustering to provide scalability and high-availability.

Application Servers, such as BEA WebLogic Server™ [4], evolved from Distributed TP Monitors in the 1990s to meet new demands imposed by the Internet. Significant among these demands is support for loosely-coupled clients, which communicate using vendor-neutral, industry-standard protocols such as HTTP [5] and SOAP [6]. Typical uses for Application Servers include:

- *E-commerce* Catalog browsing and purchasing of consumer goods such as books and appliances
- *News Portals* Personalized consolidation of news from multiple sources
- *Financial Management* Control of financial holdings such as bank accounts or stocks
- *Packaged Applications* Single-purpose, vertical applications such as accounting, expense reporting, ERP, or CRM
- *Messaging Hub* Infrastructure for routing, transforming, and managing asynchronous messages

As they have evolved, TP Systems have become increasingly distributed in two dimensions within the data center: forward, from data-centric applications in the back-end to presentation-oriented applications in the front-end, and sideways, from stand-alone »stovepipe« applications to integrative applications that mediate access to many stovepipes. To continue providing desired levels of performance, scalability, and availability at an acceptable overall system price, TP Systems have increasingly maintain data outside of centralized databases in the back-end. This often entails relaxing ACID properties of transactions within applications [7].

This paper surveys data management techniques used in Application Servers. It identifies three basic types of clustered service – stateless, cached, and singleton – that differ in the way they manage data in memory and on disk. These service types provide different ways of relaxing ACID properties of transactions. As an illustration, this paper discusses how BEA WebLogic Server uses these service types to implement the Java™ 2 Enterprise Edition (J2EE™) [8], the Java™ platform for Application Servers. The J2EE supports a variety of application programming interfaces, including ones for servlets, components, messaging, database access, and naming.

Much of the data maintained by Application Servers has its primary copy in a back-end database. However there are significant amounts of this data for which conventional databases are less than ideal. This paper outlines requirements for a persistence layer that is designed for and tightly integrated with the Application Server. This persistence layer is fundamentally distributed and takes into account issues of data replication and consistency.

This paper is organized as follows. Section 2 presents an overview of Application Server architectures. Section 3 presents the taxonomy of clustered services. Section 4 discusses the treatment of conversational state. Section 5 describes an Application Server persistence layer.

2 Application Server Architectures

Application Server systems are organized into logical tiers, each of which may contain multiple servers or other processes, as illustrated in Figure 1. The *client tier* contains personal devices such as workstations or handheld units, embedded devices such as network appliances or office machines, or servers in other enterprise systems. Such clients may be tightly- or loosely-coupled with the Application Server. The *presentation tier* manages interactions with these clients over a variety of protocols. Processes in the presentation tier, such as Web Servers or client handlers, do not run application code. The *application tier* contains Application Servers that run application code formulated in terms of servlet, component, connector, and messaging APIs. The application tier may itself be divided, for example, into servlet and transaction tiers. The *persistence tier* provides durable storage in the form of databases and file systems. The persistence tier may also contain mainframes and other back-end systems.

A common reason to segregate servers into physical tiers is to place fire-

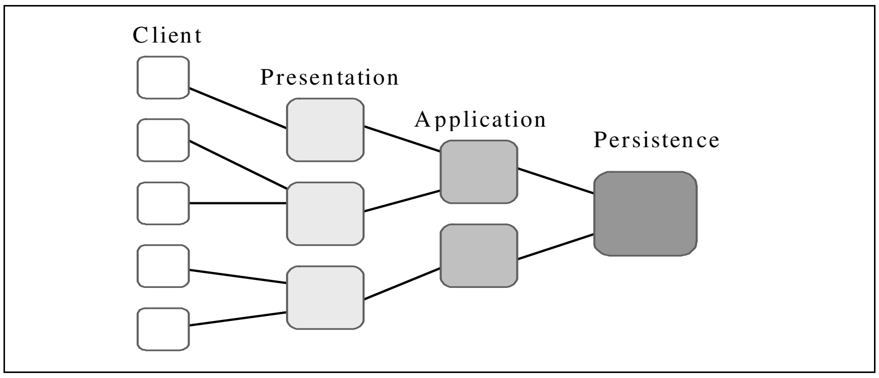

Fig. 1: Multi-Tier Cluster Architecture

walls between them. Typically, firewalls are used to protect the application tier from the outside world, but may also be used to restrict internal access to the persistence tier. Another common reason for segregating server into tiers is to improve scalability by providing s*ession concentration*. The idea here is to place many smaller machines in the front end and multiplex socket connections to fewer, larger machines in the back end. In practice, session concentration is required only by systems that support many thousands of clients.

A *cluster* is a group of servers that coordinate their actions to provide scalable, highly-available services. Scalability is provided by allowing servers to be dynamically added to or removed from the cluster and by balancing the load of requests across these servers. High-availability is provided by ensuring that there is no single point of failure in the cluster and by migrating work off of failed servers. Ideally, a cluster offers a *single system image* so that clients remain unaware of whether they are communicating with one or many servers [9]. A cluster may be contained in a tier or may span several tiers.

The typical transaction processing workload consists of many short-running requests. In this setting, parallelism is exploited most efficiently by processing each request on as few servers as possible, since the overhead for communication is relatively large. Consequently, it is preferable to minimize the number of physical tiers in the system, up to constraints such as firewalls, and to process a request on only one server in each tier. In addition, simple round robin or random load balancing schemes are particularly effective and it is rarely worth the effort either to take actual server load into account or to redistribute on-going work when it occasionally becomes unbalanced. This is in contrast to practices commonly employed for compute-intensive applications [10].

Tightly-coupled clients contain code from the Application Server and communicate with it using proprietary protocols. For this reason, they generally offer higher functionality and better performance. Load balancing and failover for such clients is built into the Application Server infrastructure. For example, WebLogic Server integrates this functionality into RMI, the basic Java API for remotely invoking methods of an object.

Loosely-coupled clients, consisting of applet-free browsers and Web Services clients, do not contain code from the Application Server and communicate with it using vendor-neutral, industry-standard protocols such as HTTP and SOAP. They generally tolerate a wider variety of evolutionary changes to the server side of an application and are easier to maintain. Load balancing and failover for such clients must be performed by external IP-based mechanisms, of which two are common. The first approach co-lists the front-end servers under a single DNS name and allows the client to make the choice. This approach provides only coarse control over load balancing and failover. Moreover, it exposes details of the system so it is both less secure and harder to reconfigure. The second approach uses a load balancing appliance that exposes a single IP address and routes to the front-end servers behind it [11].

The choice of whether to use tightly- or loosely-coupled clients is affected not only by the degree of distribution of the system, but also by the extent to which it has a centralized administrative authority. For example, a chief architect might create a collection of tightly-coupled clusters across multiple outlets of a retail company or branches of a bank or post office. Similarly, an architectural committee that spans multiple departments in the same enterprise or close trading partners in different enterprises might mandate the use of a vendor-specific messaging bus over which XML messages flow.

Along with »systematic« applications, which are carefully planned and rolled out over a long period of time, Application Servers must handle »opportunistic« applications, which are rolled out quickly and modified often during their lifetimes. As a result, Application Servers greatly benefit by the ability to tune themselves, e.g., by automatically setting thread pool and cache sizes. In addition, Application Servers must handle traffic from unknown numbers of loosely-coupled clients across the Internet. As a result, Application Servers greatly benefit from the ability to dynamically enlist resources to handle peak loads [12]. Application Servers can be integrated into an enterprise-wide Grid Computing infrastructure that facilitate sharing of resources [13]. Such features can significantly reduce total cost of ownership and increase reliability by reducing the opportunity for operator errors during reconfiguration.

3 Clustered Services

This section describes three basic types of clustered services that are used in Application Server. *Stateless services* do not maintain state between invocations. *Cached services* maintain state between invocations but keep it only loosely synchronized with the primary copy of the data in a back-end database. A stateless or cached service can be made scalable and highly available in a cluster by offering many instances of it, *any one of which is as good as any other.* Clients of the service are then free to switch between the instances as needed for load balancing and failover. This clustering technique is inappropriate for services that maintain data with strong consistency requirements, since the need to synchronize multiple copies of the data would limit both the size and the degree of distribution of the cluster. Such data should be maintained by at most one instance of a service, called a *singleton service*. If an instance of a singleton service fails, ownership of its data must be migrated to another server.

3.1 Stateless Services

Stateless services do not maintain state between invocations. They may load state from a database or other back-end system into memory, but only for the duration of an individual invocation. Stateless services may maintain non-application state, such as aggregate monitoring information or connections to back-end systems.

Application Servers provide a variety of stateless services including component APIs, such as EJB stateless session beans, and object factories, such as EJB Homes and JDBC connection pools. WebLogic Server implements clustered stateless services as follows. Using a lightweight multicast protocol, each member of the cluster advertises the instances of the stateless services it offers. The WebLogic RMI stub for a service obtains this information and uses it to make load balancing and failover decisions. The default load balancing algorithm uses a modified round-robin scheme that prefers local instances of the service, servers already involved in the transaction, and servers already connected to the client. These optimizations minimize the spread of a transaction. The default failover algorithm retries a failed operation only if it can be guaranteed that there were no side-effects, for example, because the request did not leave the server or the operation was declared to be idempotent.

3.2 Cached Services

Cached services maintain state between invocations but keep it only loosely synchronized with the primary copy of the data in a database or other back-end system. The cached data may be a direct copy of the back-end data or it may be the result of application-level processing of the back-end data. For example, the cache might contain relational rows or those rows might first be transformed into objects, HTML, or XML. In the case where the data is copied from the back-end, it may be updated by the service and written through to the database. To prevent update anomalies, the service must protect the data using optimistic concurrency control. Cached data may be written to a local disk to avoid reacquiring it after a restart and/or to recover memory.

In *on-demand* caching, values are loaded when they become needed and *evicted* when they become stale. Values may also be evicted to recover memory, a process that should be integrated with server-wide memory management. On-demand caching is appropriate for large data collections with small working sets, such as customer profile information. In *materialized* caching, values are pre-loaded during initialization and *refreshed* when they become stale. Values may not be evicted to recover memory. Materialized caching is appropriate for moderately-sized data collections that are frequently used, such as product catalogues. Since the set of data in memory is known at all times, this technique facilitates querying through the cache.

Both on-demand and materialized caching are amenable to *partitioning* [14], where responsibility for subsets of the data is striped across subsets of the servers in the cluster. Partitioning makes it possible to scale up the effective memory size of the cluster so it can manage larger data collections. Partitioning requires data-dependent routing to forward requests to the appropriate servers. In contrast, without partitioning, data accesses always occur on the local server.

In a *two-tier caching* scheme, a first-tier cached service draws its data from a second-tier cached service, which then draws its data from a back-end database as usual. Commonly, applications run in the first tier using small on-demand caches that draw data from large materialized caches, possibly partitioned, in the second tier. This architecture is often used to reduce the load on back-end databases, since it allows a single lookup to be shared by many members of the cluster. In addition, it separates heavy-duty application garbage collection from heavy-duty caching, which interact poorly. The second tier may also contain singleton services, providing a master-slave replication architecture.

Cached values may be assigned a time-to-live until eviction or refresh. This approach does not require any communication between servers, so it scales well, but requires that the application tolerate a given window of staleness and inconsistency. This approach is attractive when the data is frequently updated, e.g., from a real-time data stream, in which case keeping up with the changes can be less efficient than not caching at all. Alternatively or in addition, values may be evicted or refreshed when updates occur. This approach is attractive when the data is infrequently updated, in which case the signalling overhead will be tolerable. Update signals may be sent with varying degrees of reliability, e.g., from best effort multicast to durable messaging.

Sending update signals requires identifying when relevant back-end data has changed. This is straight-forward if updates go through the Application Server itself. However if updates go through the »backdoor«, meaning other applications that share the data, then mechanisms such as database triggers or log-sniffing must be used. For cached data that has been computed, sending update signals also requires identifying which pieces of back-end data are relevant. There is a trade off here associated with the granularity of tracking of the data: finer granularity results in longer caching but is harder to implement efficiently. If the associated queries are known in advance, then database view maintenance techniques [15] can be used. This problem is compounded in the presence of ad-hoc queries, particularly if application-level processing of the back-end data makes it unclear which queries are relevant.

WebLogic Server caches the HTML results of JSPs at either the whole page or fragment level. Fragment-level caching is useful when components of a page may be personalized for different users. A page or fragment may be tagged as being for an individual user or a group of users. Each page or fragment may be assigned a time-to-live, after which it is evicted from the cache.

WebLogic Server provides a range of caching options for EJB entity beans. An entity bean may be given a time-to-live in memory after it is loaded, during which it can be freely used to satisfy read requests in subsequent transactions. The EJB container can also be configured to send out a bean-level cache eviction signal using a light-weight multicast protocol. An instance of the container sends this signal automatically after it commits a transaction that contains updates. In addition, an API is provided to allow application code to trigger evictions manually, e.g., in the event that the application observes a backdoor update.

WebLogic Server also provides an option to keep cached entity beans consistent with the back-end database using optimistic concurrency control, but only for transactions that include writes. In addition to being used across transactions, this option can be used within a single transaction to increase database concurrency, since no database locks are held. In either case, during a transaction, the container keeps track of the initial values of certain fields, either application-level version fields or actual data fields. At commit time, these values are compared with those in the database using an additional WHERE clause in the UPDATE statement, and a concurrency exception is thrown if they don't match. The container then sends a bean-level cache eviction signal to minimize the likelihood of subsequent concurrency exceptions. Overall, although this approach does not ensure serializability, its behavior may be desirable in that it increases concurrency in acceptable ways.

WebLogic Server also provides optimistic concurrency control for disconnected RowSets, which are table-oriented results of relational database queries. A RowSet may be serialized into binary or XML format, sent across the network to a client, updated on that client, sent back to the server, and then submitted to the database.

3.3 Singleton Services

An instance of a *singleton service* has exclusive ownership of certain data items, which it is free to maintain in memory and update without danger of anomalies. If an instance of a singleton service fails, ownership of its data must be *migrated* to another server. To enable this, the instance must keep a current copy of its data in a shared database, a shared file system, or on another server. It may also maintain transient state that can be lost or regenerated after a failure. Examples of singleton services include messaging queues, transaction managers, administrative servers, in-memory databases, lock managers, long-running client sessions such as trading desks, long-running computations such as simulations or event filters, and unique ID generators.

A large singleton service may be made scalable by partitioning it into multiple instances, each of which handles a different slice of the data and the associated requests. For example, WebLogic Server allows a logical messaging queue to be partitioned into many physical instances, each of which is responsible for certain messaging consumers. In this particular case, partitioning also improves availability in that messages can continue to flow through the system after an instance of the queue has failed, although certain messages or users may be stalled until recovery occurs. Partitioning is not always appropriate in that it may result in individual requests being processed on different servers, so there is a loss of colocality. Moreover, it may not even be possible to arrange because there are no natural places to create the partitions.

A request for a singleton service must be *routed* to the appropriate server. Routing must take into account partitioning as well as migration. Routing is generally straight-forward for tightly-coupled clients, since the Application Server owns both sides of the connection. Routing for loosely-coupled clients is discussed in the next section.

There are several approaches to implementing migration of singleton service data. In *server migration*, a singleton service instance is pinned to a particular server and that server is migrated to a new machine as failures occur. The IP addresses of the server are usually migrated along with it so external references do not need to be adjusted. One advantage of server migration is that it does not require any special effort on the part of the singleton service implementer; even pre-existing services can be made highly available without modification. In addition, it is compatible with the design of most HA Frameworks [16], which offer whole-process migration.

A disadvantage of server migration is that it requires the administrator to manually partition the set of singleton service instances across a fixed set of servers. Moreover, in order for the cluster to provide automatic load balancing in the event that machines are added or removed, »over-partitioning« onto an unnecessarily large number of servers must occur. Another disadvantage of server migration is that it may take a long time to start a server and initialize the application, increasing the downtime of the service. This problem can be mitigated using hot-standby techniques, which entail implementing some kind of server/service lifecycle API.

In *data migration*, ownership of data elements is distributed and migrated among existing singleton service instances, as illustrated by the following examples.

- Each server has one Transaction Manager and failure of a server entails migrating its outstanding transactions to the Transaction Manager on another server.
- Each server has one physical instance of a logical messaging queue and failure of a server entails migrating its outstanding messages to the physical instance on another server.
- Each server has one Web container and failure of a server entails redistributing its browser sessions and Web Service conversations to the Web containers on other servers.
- Each server has one instance of an in-memory database that is in charge of some slices of the data and failure of a server entails redistributing those slices among instances on other servers.

Note that in the first three cases here, any instance of the service is as good as any other until some kind of session or connection is created, after which a particular instance must be used. The advantage of data migration is that partitioning and load balancing can be performed automatically without intervention on the part of the administrator. The disadvantage is that it considerably complicates the task of writing a singleton service.

In *service migration*, complete singleton service instances are distributed and migrated among existing servers. This approach also requires the administrator to manually partition the work, although the granularity is smaller so over-partitioning is less of an issue. A more serious problem is that this approach allows there to be multiple instances of the same service on the same server. This behavior is not acceptable for certain services such as transaction managers.

The migration problem is a classic one in distributed computing. It is hard to solve because, in an asynchronous network, there is no way to distinguish actual process failure from temporary process freezing or network partitioning [17]. Thus, a seemingly-failed process may reappear after migration has occurred, resulting in two process that think they own the same data. The standard solution is to have processes periodically prove their health to the rest of the cluster and, if that fails, shut themselves down. Processes may prove their health using distributed agreement protocols [18] and/or by accessing a shared disk. The clustering infrastructure must postpone migration for at least one health-check period to give a server the chance to shut itself down. Thus this approach introduces a natural trade-off between failover speed and safety. On a tightly-coupled hardware platform, this problem can be avoided by physically isolating the process at the level of the network switch.

WebLogic Server supports server migration by providing health checking and lifecycle APIs. These APIs allow a server to be placed under the control of WebLogic monitoring processes or an HA framework. Individual containers within WebLogic Server support workload migration as discussed in more detail in the next section.

4 Managing Conversational State

Conversational state provides a context for interpreting a sequence of related requests. Examples of conversational state include servlet session state and stateful Web Services. Conversational state can be kept in a shared database and accessed through a stateless service each time a request arrives. Alternatively, to improve performance and scalability, conversational state can be maintained in memory. In this case, it becomes the data owned by a singleton service, such as a Web container, and is migrated in the event of failure. If full durability is required, the data must be written through to the database, resulting in improved efficiency only for reads. If durability can be relaxed, the data can be replicated to a passive secondary on another server or simply lost in the event of failure.

Selection of the host for a conversation should occur when it is initially created and subsequent requests should be routed to the chosen server. For loosely-coupled clients, routing should ideally occur on the way into the cluster to avoid unnecessary network hops. Routing can be performed by external IP-based mechanisms, either by relying on a client to stick with the first server it obtains from DNS or by configuring a load balancer to offer *session affinity.* Alternatively, routing can be provided by Application Server code that resides in the presentation tier, as either a full client-handling process, such as a Web Server, or a plug-in for such a processes.

A *multi-party* conversation, such as a stateful Web Service, may involve an arbitrary number of participants. Migration of such conversations requires careful tracking of cluster membership, as described in the previous section, otherwise the participants might not agree as to which server owns the data. However for *two-party* conversations, such as servlet sessions, each participant can drive migration on its own behalf. This considerably reduces the complexity of the implementation, as described in the next subsection.

4.1 Servlet Session State

When a servlet session is first created, the hosting server generally embeds its identity in a cookie that is returned to the client and included with each subsequent request. This information can be used by the clustering infrastructure to route the request to the appropriate server.

WebLogic Server supports replication of a session to a passive secondary. Requests are handled by the primary, which synchronously transmits a delta for any updates to the secondary before returning the response to the client. To support failover, the identity of both the primary and secondary are embedded in the cookie. Figure 2 illustrates the case where the Web Server or its plug-in inspects the cookie and routes to the primary. If the primary is not reachable, it routes to the secondary, which then becomes the primary, creates a new secondary, and rewrites the cookie.

Figure 3 illustrates the case where routing is performed externally. The primary is initially created on the server where affinity has been set up. If the primary becomes unreachable, the external mechanisms switch affinity to some arbitrary member of the cluster. When the first request arrives there the servlet engine inspects the cookie, contacts the secondary to obtain a copy of the state, becomes the primary, and then rewrites the cookie leaving the secondary unchanged.

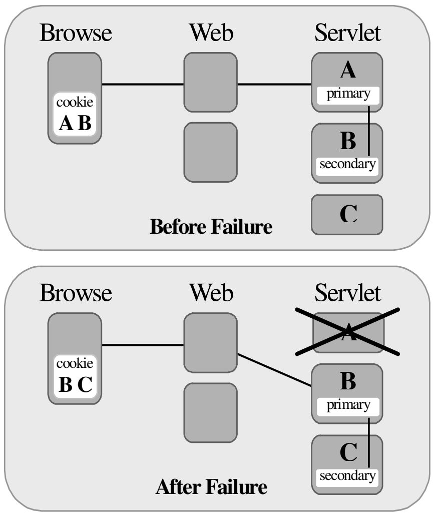

Fig. 2: Replication with Routing in the Web Server

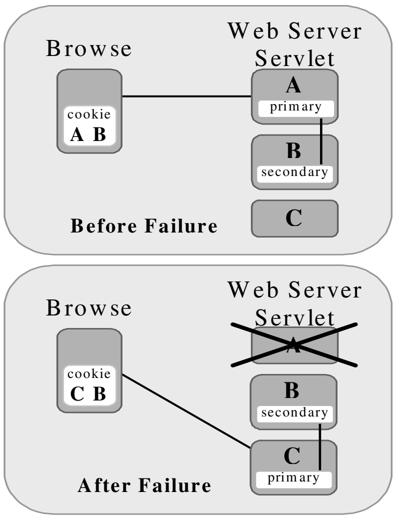

Fig. 3: Replication with External Routing

In both cases, establishment of a new primary and secondary after a failure are delayed until a new request arrives. This approach distributes the recovery work over time without reducing availability, since the new pair is effectively useless until the cookie can be rewritten.

WebLogic uses a sophisticated algorithm to place secondaries in the cluster. As part of the configuration of a cluster, it is possible to tag servers as being in named replication groups and to specify preferred groups to use for hosting secondaries. This allows the administrator to favor replication pairs being either independent, e.g., on different power grids, or dependent, e.g., on the same high-speed LAN.

4.2 Stateful Web Services

Web Services enable loosely-coupled communication between servers. Server-to-server communication can be used, for example, to implement business workflows between departments or enterprises. A fundamental characteristic of the server-to-server programming model is that it engenders multi-party conversations. Moreover, since any participant can send a request to other participants as part of a conversation, all of them must maintain state on its behalf. These characteristics are illustrated in Figure 4. Suppose client A starts a pair-wise conversation with server B, which then acts as a client and starts a subordinate pair-wise conversation with server C. On B, a single piece of state must be associated with both conversations and thus is effectively multi-party.

Another fundamental characteristic of the server-to-server programming model is that it uses asynchronous communication. In particular, store-and-forward messaging provides an attractive way of buffering work to handle temporarily disconnected or overloaded systems. In addition, store-and-forward messaging can be made reliable using simple ACKing protocols that are appropriate even for loosely-coupled systems. The alternative, transactional RPCs, is less attractive not only because the wire protocols are more complicated, but because it tightly couples resources on both sides. Note that store-and-forward messaging is distinct from client/server messaging, where producer and consumer clients interact with a central server using transactional RPCs. For store-and-forward messaging, the consumer of a message is often a process on the server itself.

Nominally, Web Service conversations should be kept in a shared database and accessed through a stateless service each time a request arrives. However Web Service conversations that are short-lived or that have bursty access patterns are attractive to maintain in memory under the management of a singleton service. Depending on the durability requirements, the data can be written through to the database, replicated to a passive secondary, or lost on failure. The latter two alternatives might be acceptable for read-only applications, shopping-cart-style applications where only the last fulfilment step is crucial, and forwarding applications where reliability is provided by the external end-points.

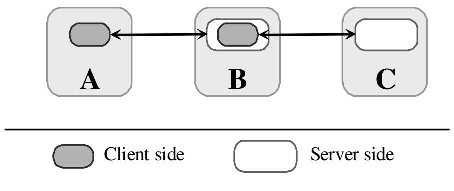

Fig. 4: Subordinate Web Service Conversations

If a Web Service conversation is maintained in memory, then any associated in-bound or out-bound asynchronous messages can be co-located with it. Assuming it is prohibitive to maintain a queue per conversation, this behavior can be accomplished by partitioning a single logical queue into one physical instance per server, each of which is responsible for certain conversations. Again, depending on durability requirements, messages can be written through to the database, replicated to a passive secondary, or lost on failure. Treating a conversation and its messages in the same way provides a nice unit of failure and failover.

The difficulty of migrating multi-party conversations is illustrated in Figure 4. If A or C were to drive migration decisions about B in isolation, then there would be disagreement as to which server owns the data. Instead, there must be cluster-wide agreement about migration and servers that are deemed to have failed must be shut down. Agreement can be controlled by a singleton service *lock manager* that grants servers the right to host conversations.

A conversational Web Service may be invoked over multiple protocols including HTTP, FTP, and SMTP. In addition, for asynchronous communication, vendor-specific binary messaging protocols may be used. In all cases, the infrastructure must perform routing, including support for both partitioning and migration.

The HTTP protocol is of particular importance here and deserves further discussion. The current HTTP-based Internet infrastructure sets up session affinity on the first request going into a cluster, but never on responses. Thus affinity will be set up for requests going from a client into a server but not in the other direction. In particular, affinity will be set up the first time a request is sent in the other direction, and the chosen server may not match the location chosen by the client. Enhancements to load balancers may eventually handle this case. The other alternative is to embed the location of a conversation in its ID, the Web Service equivalent of servlet session cookies. It is possible that standards for managing Web Service conversations will eventually support the notion of a general-purpose "biscuit" that each side is expected to echo to the other. Short of this, location embedding will be possible only at the point the conversation ID is created, which will generally occur on the client. These two techniques can be used together to solve the problem: session affinity can be used to reach conversations on the server and conversation IDs can be used to locate and route to conversations on the client.

While they can be implemented in terms of the J2EE, Web Services can benefit greatly from special packaging and optimizations, many of which are provided by BEA WebLogic Workshop™ [19]. The J2EE has radically different models for synchronous programming (typed EJB) and asynchronous programming (untyped JMS). It is more convenient for queuing to occur under the covers for void-return methods of beans. It would also be convenient to provide a special bean variant for in-memory conversations. Such conversation beans should act partly like stateful session beans, in that they are kept in memory and paged out as needed, and partly like entity beans, in that they have transactional internal state and may be shared by multiple users.

5 An Application Server Persistence Layer

As part of implementing cached and singleton services, Application Servers persist significant amounts of data for which conventional relational databases are less than ideal. This data is often in object or XML form and is accessed only in limited ways, e.g., by key or through a sequential scan. And it is often accessed only by clustered servers that coordinate their actions, obviating the need for further concurrency control. Such data is better handled by a persistence layer that is specifically designed for the Application Server. This persistence layer should be tightly integrated with the Application Server to decrease communication costs and simplify administration.

The persistent data associated with a cached service needs to be stored on a file system that is local to the server. The primary copy of such data is maintained in a centralized back-end store and the local copy is used only to speed up server initialization and make it more autonomous. The data is not transactionally updated and is kept only loosely synchronized with the primary copy. An important example here is the *metadata* needed to deploy, configure, and secure the Application Server and its applications. Local persistence should be one part of a broader metadata management solution that includes versioning and distribution.

More generally, the Application Server may cache a copy of back-end application data. This technique can isolate operational systems in the back-end from the distribution, load-handling, and error-handling requirements of presentation-oriented applications in the front-end. And the extraction, transformation, and loading process can optimize the data for the needs of these applications. For example, relational data might be pre-digested into object or XML form to avoid runtime mapping.

The persistent data associated with a singleton service needs to be stored on a shared file system to survive failure and migration. The most important example of such data is messages, both in-bound and out-bound. Gray argues that databases should be enhanced with TP-monitor-like features to handle messaging; for example, triggers and stored procedures should evolve into worker thread/process pools for servicing queue entries [20]. The counter-argument is that Application Servers should be enhanced with persistence, since they also provide much of the required infrastructure, including security, configuration, monitoring, recovery, and logging.

Specialized file-based message stores are in fact common, for all of the reasons described above, and the important point is that these stores should be opened up to include other kinds of data. Of particular importance is the state associated with long-running Web Service conversations and business workflows. Co-location of this data with its associated messages can eliminate the need to perform two-phase commit between the messaging system and the database.

6 Acknowledgements

This paper reports on work that was carried out or influenced by many talented people at BEA, in particular Juan Andrade, David Bau, Adam Bosworth, Rod Chavez, Ed Felt, Steve Felts, Eric Halpern, Anno Langen, Kyle Marvin, Adam Messinger, Prasad Peddada, Sam Pullara, Seth White, Rob Woollen, and Stephan Zachwieja. This paper is dedicated to the memory of Ed Felt.

7 References

[1] *Gray, J.; Reuter, A.:* Transaction Processing: Concepts and Techniques. Morgan Kaufmann, 1993.

[2] CICS/VS Version 1.6, General Information Manual, GC33-1055, IBM Corp., Armonk, N.Y., 1983.

[3] *Andrade, J.; Carges, M.; Dwyer, T.; Felts, S.:* The Tuxedo System – Software for Constructing and Managing Distributed Business Applications. Addison-Wesley Publishing, 1996.

[4] BEA Systems. The WebLogic Application Server. *http://www.bea.com/products/weblogic/ server/index.shtml.*

[5] Hypertext Transfer Protocol – HTTP/1.1. *http://www.ietf.org/rfc/rfc2616.txt.*

[6] Simple Object Access Protocol (SOAP) 1.1. *http://www.w3.org/TR/SOAP.*

[7] *Gray, J.:* The Transaction Concept: Virtues and Limitations. Proceedings of VLDB. Cannes, France, September 1981.

[8] Sun Microsystems. Java™ 2 Platform, Enterprise Edition (J2EE™). *http://java.sun.com/ j2ee.*

[9] *Pfister, G. F.:* In Search of Clusters, 2nd Edition. Prentice Hall, 1998.

[10] *Eager, D. L.; Lazowska, E. D.; Zahorjan, J.:* Adaptive load sharing in homogeneous distributed systems. IEEE Transactions on Software Engineering. Vol. 12, 1986.

[11] *Bourke, T.:* Server Load Balancing. O'Reilly & Associates, August 2001.

[12] *Fox, A.; Gribble, S.; Chawathe, Y.; Brewer, E.; Gauthier, P.:* Cluster-Based Scalable Network Services. Proceedings of ACM Symposium on Operating Systems Principles. Vol. 31, October 1997.

[13] *Foster, I.; Kesselman, C.; Tuecke, S.:* The Anatomy of the Grid: Enabling Scalable Virtual Organizations. International Journal of Supercomputer Applications. 2001.

[14] *Devlin, B.; Gray, J.; Laing, B.; Spix, G.:* Scalability Terminology: Farms, Clones, Partitions, and Packs: RACS and RAPS. Microsoft Technical Report MS-TR-99-85, December 1999.

[15] *Gupta, A.; Mumick, I. S. (Eds):* Materialized Views: Techniques, Implementations, and Applications. The MIT Press, 1999.

[16] *Marcus, E.; Stern, H.:* Blueprints for High Availability: Designing Resilient Distributed Systems. Wiley, January 2000.

[17] *Lynch, N. A.:* Distributed Algorithms. Morgan Kaufmann, San Francisco, 1996.

[18] *Lampson, B.:* How to Build a Highly Available System Using Consensus. In: Distributed Algorithms, Lecture Notes in Computer Science 1151, (ed. Babaoglu and Marzullo), Springer-Verlag, 1996.

[19] BEA Systems. WebLogic Workshop. *http:// www.bea.com/products/weblogic/workshop/ index.shtml.*

[20] *Gray, J.:* Queues are Databases. Proceedings 7th High Performance Transaction Processing Workshop. Asilomar CA, September 1995.

Dean Jacobs received a Ph.D. in Computer Science from Cornell University in 1985 and has served on the Computer Science faculty at the University of Southern California. In the industrial setting, he has been instrumental in the development of several commercially-available distributed systems. Currently, Dr. Jacobs is an Architect at BEA Systems, where he was one of the original designers of BEA WebLogic Server.

Dr. Dean Jacobs
BEA Systems
235 Montgomery St
San Francisco, CA 94104
USA
dean@bea.com
http://www.bea.com