Distributed Computing with BEA WebLogic Server

Dean Jacobs

BEA Systems
235 Montgomery St
San Francisco, CA 94104
USA
dean@bea.com

Abstract

This paper surveys distributed computing techniques used in the implementation of BEA WebLogic Server. It discusses how application servers provide a distributed transactional infrastructure that extends outward from backend databases. The basic treatment of data is characterized in terms of four types of clustered services that differ in the way they manage state in memory and on disk. This paper also discusses how application servers support loosely-coupled clients, both at the transport level and at the higher level of server-to-server Web Services. Finally, this paper speculates about the development of a new application server persistence layer and its use in widely-distributed computing.

1. Introduction

Application servers provide a secure, transactional, and manageable environment for building enterprise applications. Application servers are fundamentally distributed systems both because they use clustering to meet enterprise scalability and availability requirements and because they integrate physically distributed elements within and across enterprises. BEA WebLogic Server™ [1] is a distributed implementation of the Java™ platform for application servers, the Java™ 2 Enterprise Edition (J2EE™) [2]. WebLogic Server supports a variety of application programming interfaces, including ones for servlets, components, messaging, database access, and naming.

This paper examines how application servers provide a distributed transactional infrastructure that extends outward from backend databases. It identifies four types of clustered services - stateless, conversational, cached, and singleton - that differ in the way they manage state in memory and on disk. By using these services appropriately, strict ACID properties of the data can be relaxed to improve application performance, scalability, and availability [3]. This paper discusses how WebLogic Server implements the J2EE in terms of these service types.

This paper also examines how application servers support loosely-coupled clients, which communicate using simple, industry-standard protocols such as HTTP [4] and SOAP [5]. At a transport level, such clients require careful treatment of front-end load balancers and Web Servers to provide routing and session management. Several configurations supported by WebLogic Server are discussed. At a higher level, such clients engender a conversational, server-to-server programming model that unifies synchronous remote procedure calls with asynchronous store-and-forward messaging. This paper discusses the formulation of this model in WSDL [6] and its implementation in BEA WebLogic Workshop™ [7].

Finally, this paper speculates about the future relationship between application servers and databases. Application servers create and manage significant amounts of data for which conventional relational databases are less than ideal. This paper argues for the development of a new persistence layer that is tightly integrated with the application server. This persistence layer should be fundamentally distributed and should take into account issues of data replication and consistency. This paper further argues that, to increase the acceptance of widely-distributed computing models within the enterprise, application servers should be given their own copy of backend data in the manner of data warehouses. This approach isolates the operational system from the load- and error-handling requirements of widely-distributed applications and can eliminate the overhead of run-time data mapping, e.g., from relations to objects or XML.

This paper is organized as follows. Section 2 presents an overview of multi-tier cluster architectures for enterprise computing systems and compares application servers to their predecessors, distributed TP Monitors. Sec-

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment
Proceedings of the 2003 CIDR Conference

tion 3 characterizes application server treatment of data in terms of the four types of clustered services. Section 4 discusses ways in which the basic application server infrastructure needs to be enhanced to support the server-to-server programming model. Section 5 speculates about the new application server persistence layer and its use in widely-distributed computing.

2. Enterprise Computing Systems

Enterprise computing systems are primarily used for **transaction processing**, which entails fielding client requests and coordinating their submission to backend databases and other transactional subsystems. Typical transaction processing applications, such as those for banking, transportation, and manufacturing, perform simple data entry and retrieval. In contrast, typical non-transactional applications, such as those for analytical processing [8] and scientific computing, are more compute-intensive.

2.1 Multi-tier Cluster Architectures

Enterprise computing 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. The **presentation tier** manages interactions with these clients over a variety of protocols. Processes in the presentation tier, such as Web Servers, 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 a servlet tier and a transaction tier. The **persistence tier** provides durable storage in the form of databases and file systems.

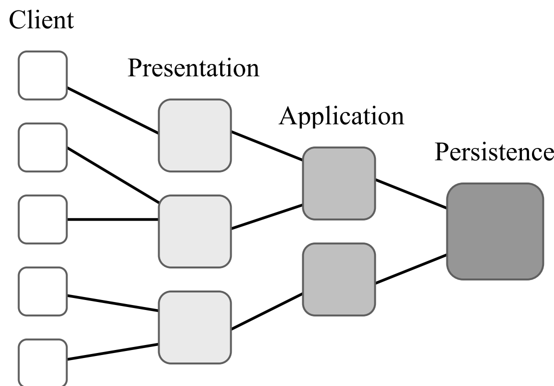

Figure 1 Multi-Tier Cluster Architecture

A common reason to actually separate servers into physical tiers is to place firewalls between them. Firewalls are typically 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 **session 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. The limit here occurs at the persistence tier, where a small number of powerful machines provide a reliable, shared foundation for the rest of the system. In practice, session concentration in the front end is required only by systems that support tens of 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 standard transaction-oriented 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].

One limitation to the scalability of an enterprise computing system is its ability to concentrate data in an individual place, such as a backend database or the memory of a server. It is often possible to mitigate this problem by **partitioning** the data so that different elements are concentrated in different components of the system [11]. Partitioning requires data-dependent routing to ensure that each request is handled by the appropriate component. For the transaction-oriented architecture described above, such routing takes the form of data-dependent load balancing upon entry to a physical tier.

2.2 Tightly- and Loosely-Coupled Clients

The clients of an application server - personal devices, network appliances, or servers in other enterprise computing systems - may be tightly or loosely coupled with it.

Tightly-coupled clients contain code from the application server and communicate with it using proprietary protocols. For this reason, they generally offer more functionality and better performance. Load balancing and failover for such clients is built into the application server infrastructure.

WebLogic Server integrates load balancing and failover for tightly-coupled clients into its implementation of RMI, the basic Java API for remotely invoking methods of an object. The WebLogic RMI stub for a service obtains information about which members of the cluster are actively offering the service and uses it to make load balancing and failover decisions. The algorithm for obtaining this information and making these decisions is pluggable and is deployed along with the service.

WebLogic Server distinguishes between internal and external tightly-coupled clients. **Internal** clients are services that invoke other services from inside the application tier. They obtain the information necessary to perform load-balancing and failover from the local server. **External** clients run in a client-side environment that may be more independently administered. They occasionally contact a member of the cluster to obtain load-balancing and failover information and cache it locally.

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 only simple, industry-standard protocols such as HTTP and SOAP. Such clients tolerate a wider variety of evolutionary changes to the server side of an application and are therefore easier to maintain. In addition, they may be developed more independently and are therefore better to use when crossing lines of administrative authority.

For loosely-coupled clients, load balancing and failover 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 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 [12]. Such appliances can perform sophisticated forms of load balancing and failover, e.g., to support partitioning.

Note that tightly- and loosely-coupled clients use different protocols and therefore invoke services using different APIs.

2.3 Transaction Processing Monitors

Transaction processing monitors, or **TP monitors**, were developed in the 1980s to provide an environment for building transactional applications [13]. Distributed TP monitors, such as BEA Tuxedo™ [14], run on a cluster of mid-sized server machines rather than a mainframe. Distributed TP monitors have evolved into application servers as they are today largely in order to meet new demands imposed by the Internet and by programming languages based on virtual machines.

TP monitors were originally designed to work with tightly-coupled clients. Distributed TP monitors generally provide special processes in the presentation tier for this purpose. Tuxedo provides both **workstation handlers**, which route requests from workstation clients to servers in the application tier, and **server gateways**, which manage interactions between Tuxedo systems. Server gateways control the import and export of services, concentrate traffic to improve scalability, and provide a locus for interposed transactions. Application servers could usefully provide such features, however the expectation nowadays is that most interactions with remote systems will be loosely-coupled.

All of the widely-used application servers today are built on virtual machines such as the Java™ VM. The resulting systems are generally constructed out of small numbers of large, homogeneous processes. Since it is usually not safe to restart individual failed services within a VM, such systems have a coarse granularity for failure. In contrast, TP monitors like Tuxedo are constructed out of large numbers of small, heterogeneous processes, such as workstation handlers and server gateways, and have a smaller granularity for failure. In particular, Tuxedo maintains its execute queues in processes that do not contain user code, decreasing the likelihood that failures will result in the loss of pending requests. Technological advances to decrease the overhead of incrementally adding VMs to a machine or to provide isolation within a VM will improve the reliability of application servers.

TP Monitors are for the most part statically configured, in keeping with a focus on stability and predictability. This approach is suitable for "systematic" applications, which are carefully planned and rolled out. The associated workloads are fairly steady, or at least well-known, and peak loads can be handled by over-provisioning together with an internal server policy of "deny rather than degrade service". The total cost of setting up and tuning such a system may be high, but this is offset by the fact that it will be in operation with only small modifications for a long period of time.

Along with systematic applications, application servers must handle "opportunistic" applications, which are rolled out quickly and modified often during their lifetimes. In addition, application servers must handle traffic from unknown numbers of loosely-coupled clients across the Internet. As a result, application servers have a greater need to be self-tuning and to dynamically enlist computing resources to handle peak loads [15]. 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 characterizes basic application server treatment of data in terms of four types of clustered services - stateless, conversational, cached, and singleton - that differ in the way they manage state in memory and on disk. It describes the way in which WebLogic Server implements certain J2EE APIs in terms of these services.

3.1 Stateless Services

A **stateless service** does not maintain state in memory between invocations. It may load state from a persistent store into memory, but only for the duration of an individual invocation. A stateless service can be made scalable and highly-available simply by deploying multiple instances of it in a cluster. Load balancing and failover between these instances is straight-forward because any one of them is as good as any other.

For tightly-coupled clients, application servers provide explicit stateless component APIs, such as EJB stateless session beans. They also provide stateless factories for accessing stateful components and connectors, such as EJB Entity Beans, JDBC database connections, and JMS messaging system connections. Finally, stateful components such as EJB Entity Beans can be implemented in a stateless manner by writing out the state to shared storage between invocations.

WebLogic Server implements clustered stateless services for tightly-coupled clients as follows. Recall from section 2.2 that the RMI stub for a service obtains information about which members of the cluster are actively offering the service and uses it to make load balancing and failover decisions. In the case of stateless services, the members of the cluster disseminate this information using a lightweight multicast protocol. To choose a server, the default load balancing algorithm uses a round-robin scheme with several important extensions. First, and this applies only to internal clients, the algorithm always prefers a local instance of the service in order to minimize the number of servers involved in processing a request. Second, if a local instance is not available but a transaction is in progress, the algorithm gives preference to servers that are already involved so as to limit the spread of the transaction. Finally, the default algorithm retries a failed operation only if it can be guaranteed that the operation did not have side-effects, for example, because the request did not leave the server or the operation was declared to be idempotent.

For loosely-coupled clients, the basic APIs are either stateless or can be implemented in a stateless manner by writing out internal state to shared storage between invocations. These options apply to servlets as well as Web Services. Load balancing and failover may be performed by external IP-based mechanisms, as described in Section 2.2, or by application server code that resides in the presentation tier. These issues are discussed in more detail in the context of conversational services in Section 3.2.

3.2 Conversational Services

A **conversational service** is an instance that is earmarked for processing only and all requests from a particular client within a session. Conversational services maintain state in memory, and this state is generally lost in the event of failure. Conversational state may be paged out on an as-needed basis to free up memory. Performance need not be impacted in this case because updates are not expected to be individually written to the disk and the data is not expected to survive failures. Conversational services can be made scalable by distributing their instances across a cluster; the application server infrastructure must route all requests within a session to the appropriate instance.

For tightly-coupled clients, application servers provide explicit conversational component APIs, such as EJB stateful session beans. In the WebLogic Server implementation of this API, load balancing occurs when a (stateless) EJB home is chosen to create a stateful session bean. The associated RMI stub is hardwired to the chosen server so requests are naturally routed to the right place.

To improve the availability of stateful session beans, WebLogic Server offers primary/secondary replication with the secondary instances also distributed across the cluster. The stub keeps track of the secondary as well as the primary and performs failover as required. The primary sends update deltas to the secondary on transaction boundaries, a scheme originally developed for the Tandem NonStop Kernel's process pairs [16]. Its use in this setting creates some anomalies because the internal state of a stateful session bean is not transactional, thus failure of the primary can result in unexpected roll back upon failover to the secondary. Customers universally prefer this behavior to the more expensive option of sending deltas on every update.

For loosely-coupled clients, requests coming into an application server are often grouped into sessions. This practice applies to browser clients, which engage in browser sessions, as well as Web Service clients, which engage in conversations. A session is generally associated with a piece of state that must be maintained between requests: browser sessions are associated with servlet session state and Web Services may be stateful.

Session state may be written out to shared storage between invocations, in which case the service is stateless. If durability is not required however, then there are several alternatives that can improve performance and scalability. First, session state can be sent back and forth between the client and server under the covers, again resulting in a stateless service. This approach is not always feasible or desirable, particularly if there are large amounts of data. Second, and more commonly, session state may be left in memory on the server-side between requests, resulting in a conversational service.

Load balancing of conversational services for loosely-coupled clients requires some care because the underlying protocols assume everything is stateless. Load balancing should occur only when the session is first created and all subsequent requests should be routed to the chosen server. Such **session affinity** can be provided by external IP-based mechanisms, either by relying on a client to stick with the first server it obtains from DNS or by appropriately configuring a load balancer. Alternatively, it 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. Common practice is to have the hosting server embed its location in a session cookie that the client returns with each new request. The application server code that resides in the presentation tier inspects this cookie and then routes the request to the appropriate place. Equivalent functionality can also be provided using URL rewriting.

To improve the availability of in-memory servlet session state, WebLogic Server offers primary/secondary replication with the secondary instances also distributed across the cluster. 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 from the primary to the secondary, 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. 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.

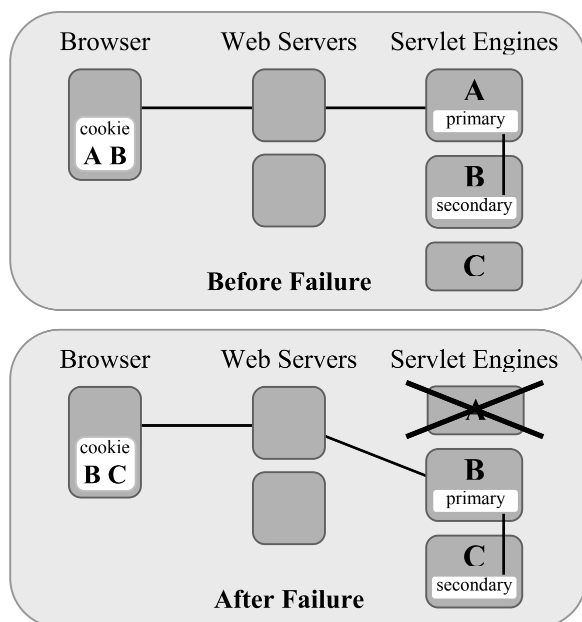

Figure 2 Replication with Routing in the Web Server

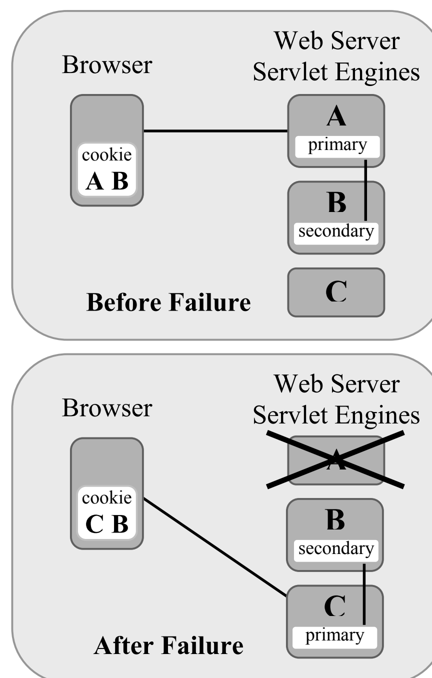

Figure 3 Replication with External Routing

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. Each server chooses one preferred server to host its secondaries. This reduces the interconnectedness of the cluster and facilitates bulk, asynchronous replication. The algorithm for choosing the secondary server organizes the candidates into a logical ring and looks for the first one in the desired replication group that is on a different machine.

The treatment of in-memory conversational state for Web Services is related to the above, however it is complicated by the peer-to-peer nature of the interaction. This issue is discussed in more detail in Section 4.

3.3 Cached Services

A **cached service** maintains data in memory and uses it to process requests from multiple clients. The cached data may be directly drawn from a backend store or it may be the result of application-level processing of backend data. For example, the cache might contain relational rows or those rows might first be transformed into objects, HTML, or XML. Updates of data from a backend store are always written to disk; the cache is used only to satisfy read requests. Cached data may be written to local disks in the middle tier so it does not need to be regenerated after a failure. Like a stateless service, a cached ser-

vice can be made scalable and highly-available simply by deploying multiple instances of it in a cluster. Load balancing and failover between these instances is straight-forward since any one of them is as good as any other.

Implementations of cached services differ in the extent to which they ensure that the copies of cached data are consistent with each other and with the backend store. Increased consistency generally comes at the expense of scalability, performance, and/or functionality, and a variety of options should be provided to meet the needs of different applications.

The simplest approach is to have each cache flush itself at regular intervals according to a configured **time-to-live** value. This does not require any communication between the servers, so it scales well, but requires that the application tolerate a given window of staleness and inconsistency. This approach is attractive when the backend data is frequently updated, e.g., from a real-time data stream, in which case keeping up with the changes would be tantamount to not caching at all. A step beyond this is to flush the caches after each update completes, but not within the updating transaction so a window of staleness and inconsistency will still exist. This approach is attractive when the backend data is infrequently updated, in which case the overhead for signalling the flushes will be insignificant.

A third approach is to keep all copies of the data consistent with the backend store using some form of concurrency control in the caches. If optimistic concurrency is used, the system should flush the caches after updates to reduce the possibility of concurrency exceptions. The use of pessimistic locking in this context is discussed in more detail in Section 3.4.

An alternative to flushing the caches is to initially preload them with specified slices of data and then to refresh the slices as updates occur. As with flushing, refreshing can occur at regular intervals, after updates but outside the transaction, or consistently with a backend store. Since the set of data in memory is known at all times, this approach facilitates querying through the cache in the manner of in-memory databases [17]. The notion of storing slices of backend data on local disks in the middle tier is discussed in Section 5.

Flush- and refresh-on-update both require identifying which pieces of data in the backend store are used to compute which pieces of data in the cache. There is a trade off here associated with the granularity of tracking of the backend 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 [18] can be used. This problem is compounded in the presence of ad-hoc queries, particularly if application-level processing of the backend data makes it unclear which queries are relevant.

Flush- or refresh-on-update also require identifying when relevant data in the backend store has been updated. This is straight-forward if the updates go through the application server itself. If the updates go through the "backdoor", meaning other applications that share the data, then either triggers or log-sniffing must be used.

Caching can occur in any tier of an enterprise computing system. As it moves closer to clients, the benefits of a cache hit increase in that round-trip times are reduced and a lighter load is placed on the backend infrastructure. On the other hand, the data may become tailored to the needs of particular clients so that it can be less-generally shared. More importantly, it becomes increasingly difficult to ensure the integrity of the data and to keep it consistent with the copy of record in the backend. For anything other than largely static files, it is probably best not to place caches beyond the firewall.

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 is assigned a time-to-live, after which it is flushed from the cache.

WebLogic Server provides a full range of consistency options for cached 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 flush signal using a light-weight multicast protocol. An instance of the container (a node in the cluster) sends this signal automatically after it commits a transaction that contains updates. In addition, an API is provided to allow application code to trigger a cache flush 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 backend store using optimistic concurrency, 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 flush 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 for disconnected RowSets, which are the 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.

3.4 Singleton Services

A **singleton service** is active on only one server in the cluster at a time and processes requests from multiple clients. A singleton service is usually backed by private, persistent data, which it caches in memory. It may also maintain transient state in memory, and this state is lost in the event of failure. The clustering infrastructure is responsible for creating and activating singleton services. After a singleton service is activated, it must establish its own internal state by accessing the backend store.

A **continuous** singleton service is active on **exactly one** server at all times. As examples, continuous singleton services can be used to implement message queues, transaction managers, and administrative servers. Upon failure, a continuous singleton service must be pro-actively either restarted on the same server or migrated to a new server. Typically, an administrator specifies a list of possible servers for a continuous singleton service and the clustering infrastructure keeps it on the most-preferred server that is currently active. Clients of a continuous singleton service access it remotely.

An **on-demand** singleton service is active on **at most one** server at a time. It may be activated on, or migrated to, the server where it is going to be used, or it may be accessed remotely. On-demand singleton services tend to be lighter in weight and greater in number than continuous singleton service. As examples, on-demand singleton services can be used to implement shared conversational services, consistently-cached persistent components such as EJB entity beans, and information about users such as profile data and message subscriber data. The ability to adjust the location of on-demand singleton services allows the clustering infrastructure to adapt to meet the demands of the workload.

A large singleton service may be made more scalable by partitioning it into multiple instances, each of which handles a different slice of the backend data and its associated requests. For example, a message queue might be partitioned along the lines of message producers or consumers [19]. 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 co-locality. Moreover, it may not even be possible to arrange because there are no natural places to create the partitions.

A group of singleton services may be aggregated into one singleton service to simplify administration and reduce bookkeeping overhead. For example, instead of implementing each entity bean as its own on-demand singleton service, the entire EJB home might be implemented as a single continuous singleton service. The overall key space might then be partitioned among several such homes to improve scalability. This approach is attractive for applications that can be end-to-end partitioned, e.g., by user ID number, so co-locality is not lost.

Implementations of singleton services must avoid the classic distributed computing problem of "split-brain syndrome" during migration. Suppose a target server establishes ownership of a singleton service and initiates some associated operation in its own thread. Then suppose the target server temporarily freezes or is isolated from the cluster, and management servers migrate ownership of the service. Even if the target server immediately notices the ownership change, there is little it can do about the on-going operation. Distributed consensus protocols [20] can ensure that other servers ignore subsequent messages from the target server, but don't prevent it from sending messages to clients or updating a database.

The general solution to this problem has the following form. The target server establishes its continuing availability with the management servers by performing some action, such as sending a heartbeat or responding to a health monitoring query, at regular intervals. If the management servers do not observe this action within some **grace period**, they (may) migrate ownership of the service. The target server attempts to ensure that all of its operations associated with the service complete within the grace period, so that split-brain does not occur even if it loses contact with the management servers. Finally, if possible, the management servers physically isolate the target server from clients and disks upon migration of the service. Such isolation greatly reduces the potential for errors if operations do not complete within the grace period. It therefore allows the grace period to be smaller, speeding up migration. Note that such isolation can be accomplished only in platform-dependent ways.

A specific practical solution to this problem might have the following two-level form. First, continuous singleton services are directly implemented using either an HA framework [21] or some kind of distributed consensus protocol. The latter approach can be used in conjunction with SNMP-based router-level fencing to provide isolation. In any case, such a solution will be fairly heavy-weight and should be used for only a handful of services. Second, these baseline services are used to bootstrap a highly-available **lease manager** which grants leases to own services [22]. Lease owners must regularly perform a handshake with the lease manager to renew their leases. In the general terms presented above, this handshake establishes the target server's availability and the lease period corresponds to the grace period. The lease manager should support "push" leases for continuous singleton services and "pull" leases for on-demand singleton services. The lease table should be persistent, so it survive failures, in order to ensure that creation of a service occurs only once. Leasing may also be used in conjunction with router-based fencing.

WebLogic Server takes a multi-faceted approach to ensuring the availability of singleton services, work that is on-going. The first line of defence is to harden individual

servers against failures. This is done, for example, by allowing each server to have multiple network adapters to guard against network failures. Second, health monitoring and lifecycle APIs are provided to allow detection and restart of failed and ailing servers. Through these APIs, a server may be placed under the control of a WebLogic node manager process or a platform-specific HA framework. Third, it is possible to do software upgrades without interrupting services; this applies to rolling upgrades of server software as well as hot redeploy of application software. Finally, it is possible to migrate singleton services. Services may be deployed into named targets, each of which is migrated as a unit so that service co-location can be maintained.

4. Cluster-to-Cluster Interactions

This section discusses interactions *between* application clusters. A fundamental issue here is the degree to which the system as a whole has a centralized architectural and administrative authority. As control becomes more centralized, it becomes more feasible for the clusters to communicate using proprietary protocols, which generally offer more functionality and better performance.

At one end of the spectrum, a single authority within an enterprise might create a collection of tightly-coupled clusters that communicate using only proprietary protocols. This configuration might exist to distribute a single application across different branches of the company or to scale up a collection of tightly-coupled applications within one branch. For this purpose, Tuxedo offers cluster-to-cluster gateways as described in section 2.3. WebLogic Server supports the notion of an administrative **domain** - the unit of startup, shutdown, configuration, and monitoring - which can contain multiple clusters. As an example, Weblogic domains might be used to set up a multi-tiered system in which each tier is its own cluster.

At the other end of the spectrum, the clusters might be in different enterprises with no coordination between them. This configuration might occur when trading partners are being dynamically discovered and linked into business processes and workflows. In this case, industry-standard Web Services protocols, such as SOAP over HTTP, are essential to ensure interoperability between the systems. SOAP uses XML to provide self-describing, extensible payloads, which make it easier to modify one system without effecting others. Perhaps more importantly, since it is low in functionality, SOAP is simple. As the protocol is extended to include more sophisticated features, such as transactions and transport independence, interoperability may well suffer.

Between these two extremes, there might be a central authority that mandates the use of certain proprietary communication technologies, such as a messaging bus, over which XML messages flow. This authority might exist between departments in the same enterprise or between close EDI-style trading partners.

Regardless of the degree of coupling, the server-to-server programming model has several unique characteristics. First, asynchronous communication is essential to support long-running business operations that flow back and forth between clusters. 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 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.

A second unique characteristic of the server-to-server programming model is the organization of work into peer-to-peer conversations. Either participant can contact the other within a conversation and both must maintain state on its behalf. In order to centralize the interface specifying the methods that may be called within a conversation, a notion of **callbacks** is required.

The server-to-server programming model is well characterized by WSDL, the Web Services Description Language. WSDL supports four types of operations, thereby providing a unified model for synchronous RPC and asynchronous messaging along with an explicit notion of callbacks.

1. **One-way** Receive a message
2. **Request-response** Receive a message and send a correlated message
3. **Solicit-response** Send a message and receive a correlated message
4. **Notification** Send a message

A server offers a WSDL service and a client (of the service) initiates a one-on-one conversation with the server. All methods invoked as part of the conversation must be named in the server's WSDL. In particular, within the conversation, the server may asynchronously contact the client using one of the specified callbacks, but not by invoking a new service on the client. The server may initiate **subordinate** conversations with other servers, including the client, but these are distinct conversations in which the server acts as a client. Note that this model is evolving and may ultimately support a higher-level notion of conversations that span multiple parties and multiple basic services.

Both the client and server sides of a conversation must maintain state on its behalf. Unlike servlet session state, Web Service conversations are generally long-lived and support (per-method, server-demarcated) transactions. More significantly, the server side of a conversation must be linked to the client side of any subordinate conversations and this must be done in a way that isolates the dif-

ferent interfaces. As an example, illustrated in Figure 4, suppose that client A has a conversation with server B, which then acts as a client in a subordinate conversation with server C. If B were to naively use the same object to handle requests for both conversations, then callbacks from C would be accessible as call-ins from A. Moreover, it would be problematic for B to create subordinate conversations with other services of the same type as C, since the source of callbacks would be ambiguous. Thus, either the object on B needs to handle callbacks differently from call-ins, or B must create a separate but dependent object for each subordinate conversation. In either case, a conversation may have several simultaneous users. This is again in contrast to servlet session state, where each session has only one user.

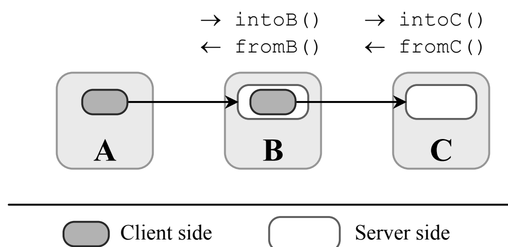

Figure 4 Subordinate Web Service Conversations and State Management

While Web Service conversations will generally be long-running and durable, there are circumstances where it is acceptable to leave them in memory. Examples here include 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. Any in-bound or out-bound asynchronous messages for an in-memory conversation should be queued in-memory along with it. This approach provides a nice unit of failure in that the conversation and its messages are lost together.

The hard part about implementing in-memory conversations is locating them within a cluster. 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 the client into the server but not for callbacks from the server to the client. In particular, affinity will be set up the first time such a callback occurs, 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 for implementing in-memory conversations 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. The miracle here is that 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.

The above approach does not allow a conversation to be moved after it has been established. Conversation migration is needed to support primary/secondary replication as well as to optimize the overall system around its most active participants. Since a conversation may have several simultaneous users, migration requires that conversations be implemented as on-demand singleton services, as discussed in section 3.4.

While they can be directly 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. The J2EE has radically different models for synchronous programming (typed EJB) and asynchronous programming (untyped JMS), which does not match well with WSDL. It is more convenient for queuing to occur under the covers for void-return methods of beans. The underlying implementation should support store-and-forward messaging as well as client/server messaging. Another useful feature is 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. Future Directions

5.1 A Middle-Tier Persistence Layer

Application servers create and manage 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. This suggests that a new persistence layer be developed with the specific needs of application servers in mind.

A crucial requirement of this new persistence layer is that it be distributed across the middle tier so data is kept close to its use. Ideally, the persistence engine should be part of the application server itself to decrease communication costs and simplify administration. These considerations argue against constructing the middle-tier persistence layer out of conventional relational databases [23], which are heavyweight and physically separate from the application server.

Messages, both in-bound and out-bound, are one of the most significant categories of middle-tier data. 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 [24]. 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 middle-tier data. Significant performance gains can be realized by having these stores include the data needed to process in-bound messages, in particular, the conversational state associated with long-running, cluster-to-cluster workflows. Co-location of this data can eliminate the need to perform two-phase commit between the messaging system and the database. Two-phase commit is otherwise required even if the messaging system keeps all of its data in the database, because the messaging system needs to be made aware of the outcome of each transaction in order to adjust its internal data structures.

Another important category of middle-tier data is the internal information needed to administer the application server. Deployment, configuration, and security information need to be distributed across servers in the cluster. Servers can start more rapidly and more autonomously if this information is stored on local disks. Monitoring, testing, tracing, and auditing logs need to be collected from servers in the cluster and integrated together.

5.2 Widely-Distributed Computing

The majority of enterprise computing systems today are integrated with the Internet only though web browsers. Application servers are used primarily to support this functionality, and they do so as relatively stand-alone "stovepipes" at the front end of the data center. Web Services are intended to make Internet technologies suitable for more fundamental business processing, tying together backend systems within and across data centers. Web Services can also provide a basis for integrating the enterprise with application and storage service providers, peer-to-peer computing technologies [25], and computational grids [26].

The acceptance of such widely-distributed computing models within the enterprise has been hampered by the requirement to tightly control access to critical business data. One possible solution is to provide a separate middle-tier copy of the backend data, in the manner of a data warehouse, for the use of widely-distributed applications. This approach leads to a multi-cluster architecture, illustrated in Figure 5, in which a remote transaction cluster and an analytics cluster are fronted by a presentation cluster and backed by an operational transaction cluster.

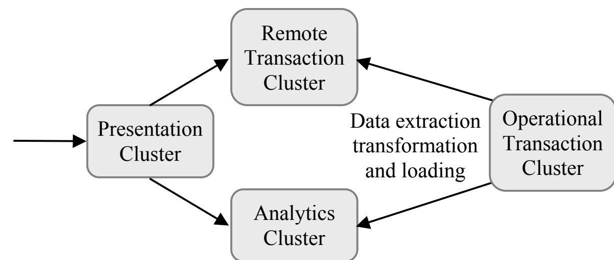

Figure 5 Multi-Cluster Architecture for Transactional and Analytic Applications

This approach isolates the operational system from the distribution, load-handling, and error-handling requirements of remote applications. 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 middle-tier copies will in general be less up to date than the copy of record in the backend, and this might require changing the way certain business processes work. For example, this approach fits naturally with the airline reservation / shopping cart model, where a series of best-effort operations lead to a single critical fulfilment step which may fail. Optimistic concurrency techniques are ideal here. Not all business processes can be formulated in these terms, but those that can have the greatest chance of being successfully distributed.

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.

References

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

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

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

[4] Hypertext Transfer Protocol -- HTTP/1.1. http://www.ietf.org/rfc/rfc2616.txt

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

[6] Web Services Description Language (WSDL) 1.1. http://www.w3.org/TR/wsdl.html.

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

[8] E. Thomsen. *OLAP Solutions: Building Multidimensional Information Systems, Second Edition*. Wiley. 2002.

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

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

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

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

[13] J. Gray, A. Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufman, 1993.

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

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

[16] J.F. Bartlett. A NonStop Kernel. *Proceedings 8th Symposium on Operating Systems Principles*. Pacific Grove, CA, December 1981.

[17] The TimesTen Team. High Performance and Scalability through Application-Tier, In-Memory Data Management. *Proceedings of VLDB*. 2000.

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

[19] S. Grant, M. P. Kovacs, M. Kunnumpurath, S. Maffeis, K. S. Morrison, G. S. Raj, P. Giotta. *Professional JMS Programming*. Wrox Press, March 2001.

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

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

[22] C. Gray, D. Cheriton. Lease: An efficient fault-tolerant mechanism for distributed file cache consistency. *Proceedings 12th ACM Symposium on Operating Systems Principles*, 1989.

[23] Q. Luo, S. Krishnamurthy, C. Mohan, H. Pirahesh, H. Woo, B. G. Lindsay, J. F. Naughton. Middle-tier Database Caching for e-Business. *SIGMOD Conference*. 2002.

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

[25] D. P. Anderson, J. Kubiatowicz. The Worldwide Computer. *Scientific American*. March 2002.

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