

Adding High Availability and Autonomic Behavior to Web Services

Ken Birman, Robbert van Renesse, Werner Vogels¹

Dept. of Computer Science, Cornell University

{ken,rvr,vogels}@cs.cornell.edu

Abstract

Rapid acceptance of the Web Services architecture promises to make it the most widely supported and popular object-oriented architecture to date. One consequence is that a wave of mission-critical Web Services applications will certainly be deployed in coming years. Yet the reliability options available within Web Services are limited in important ways. To use a term proposed by IBM, Web Services systems need to become far more “autonomic,” configuring themselves, diagnosing faults, and managing themselves. High availability applications need more attention. Moreover, the scenarios in which such issues arise often entail very large deployments, raising questions of scalability. In this paper we propose a path by which the architecture could be extended in these respects.

1. Introduction

There is a tremendous difference between a computing system that *works* and one that *works well*. In what follows, we suggest that for mission-critical purposes, the Web Services architecture risks working, but not particularly well, and that this poses serious issues for enterprises that are already deeply committed to Web Services deployments. If these two trends continue unchanged, we face a future in which important applications will lack the assurance properties that their users expect, with all sorts of undesirable implications.

A thorough treatment of assurance properties would need to consider several kinds of reliability and security issues, system installation, configuration and self-management. This paper is considerably narrower: we focus on reliability issues associated with highly available applications – applications that need to remain operational and rapidly responsive even when failures disrupt some of the nodes in a system. We suggest that

although reliability has received a great deal of attention in the W3 community, availability has been largely overlooked. Indeed, high availability applications with quick response times appear to be at odds with some aspects of the Web Services reliability model.

A decade or more in the past, such a finding might have been seen as a comment on the state of the art, but now we know far more about high availability, scalability, and other forms of reliability [1, 2]. Today, the real implication of our observation is that Web Services are missing some basic components that have proved themselves invaluable in past work on highly critical computing applications for settings such as stock markets and air traffic control systems. But Web Services systems are also expected to automatically find servers and configure themselves, and then to operate securely and reliably in a completely automated manner. This goes beyond what we know from the past, pointing to a need for a new kind of technology – what IBM has been calling “autonomic” functionality.

This paper proposes extensions to the Web Services architecture to support mission-critical applications. Examples include standard services that track the health of system components, mechanisms for integrating self-monitoring, self-diagnosis of faults and self-repair into applications, automated configuration tools, scalable event reporting mechanisms, and tools for large-scale data mining. Our extensions appear to be fully compatible with the existing framework. We are implementing the proposal and hope to have a useable platform in place by sometime in 2005.

To illustrate the issues, this paper focuses on a hypothetical scenario that might arise if an online vendor were to offer developers a library of “Web Serverlets” for inclusion into third-party applications. These developers include the serverlets into their applications, and then deploy them onto “4th party” end-user systems. Our e-tailer now faces the challenge of ensuring that the

¹ Our work was supported in part by DARPA/AFRL Grant number IFGA-F30602-99-1-0532 and in part by the National Science Foundation under its ITR program. Additional support was provided by the AFRL Infrastructure Assurance Institute, AFOSR, and Microsoft Research.

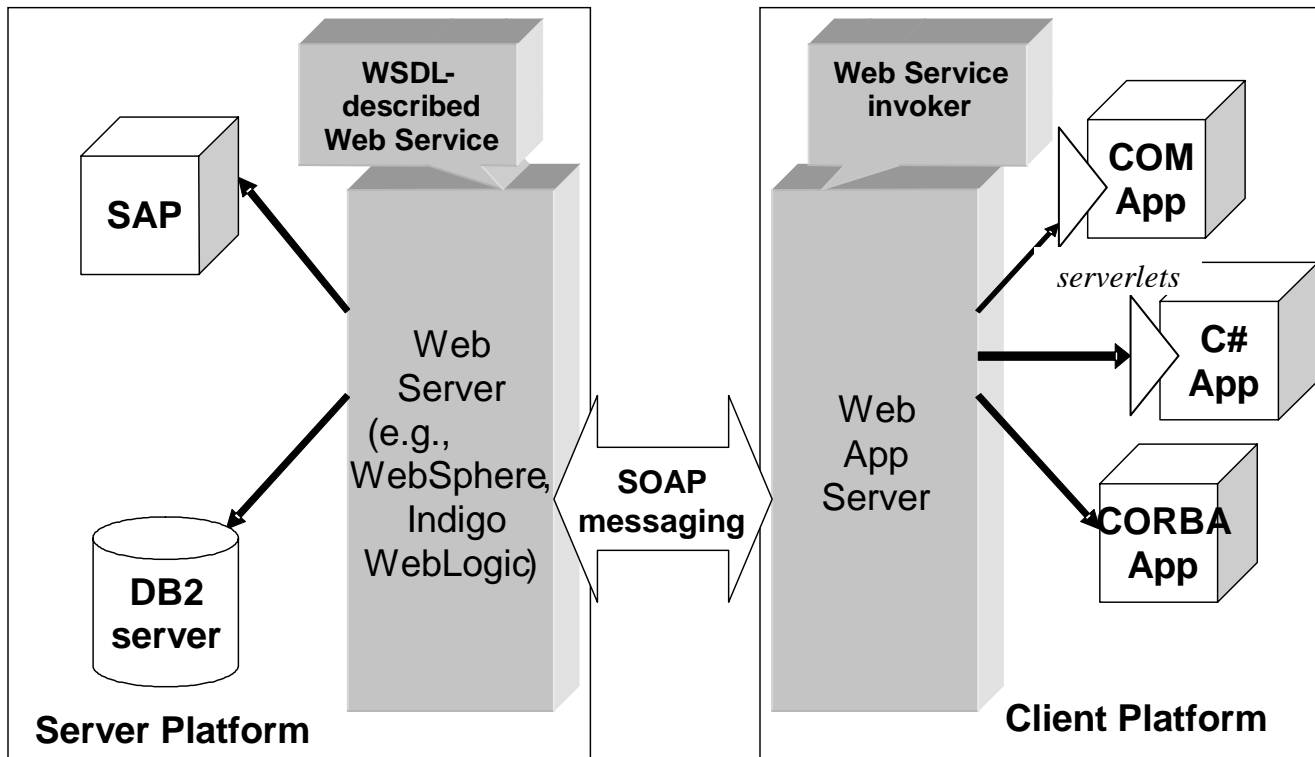


Figure 1: Web Services Client/Server Structure.

resulting system – a true 4-tier system in which there may be serverlets running on tens of thousands of end-user platforms – will operate correctly.

2. The Web Services Architecture

The Web Services architecture is generally described at two levels. The first looks at the component structure of a typical Web Services client that has located and bound itself to a Web Services server, which in turn serves as a front end for one or more back-end servers. The Web Services front end runs on one more front-end machines, and the client systems run on a diversity of remote machines, accessing the data center over the Internet. Figure 1 depicts this scenario.

A second way to discuss the Web Services architecture focuses on the behavioral abstractions supported. This leads to the sort of component stack diagram seen in Figure 2. Here, one sees a set of standards endorsed by the W3 consortium, typically describing protocols that orchestrate actions spanning both clients and servers. In this paper, we'll be particularly interested in reliability options available through the standards WS_Reliability, WS_ReliableMessaging and WS_Transactions.

3. Reliability

Although the Web Services architecture includes a reliability specification and an underlying reliable

message passing component, the details of these parts of the framework are still a subject of debate. To understand the nature of the problem, we'll now consider a scenario that might arise in our e-tailer's 4-tier system. A close look will then reveal that while the architecture should work well for certain kinds of applications, there are others that just won't fit comfortably within the proposals now on the table.

For clarity of exposition, we'll assume that our e-tailor is a medical supply vendor. The Web Serverlet library consists of a collection of small applets that can be used in client applications to place orders, track status, obtain copies of invoices, pay outstanding balances, and so forth. A typical application might be a system developed to automate a medical practice or a hospital. By including these Web Serverlets into the application, the developer is freed to concentrate on functionality needed by the end-users, tapping into the e-tailer's backend systems and order fulfillment capabilities without needing to duplicate what would presumably be an enormously complex system.

The reliability of the resulting product may boil down to a question of reliability of the e-tailer's 4 tier architecture. If something causes an operation to delay "unreasonably", the hospital may be unable to place orders for urgently needed supplies, and will soon be on the telephone to the application developer, who will turn

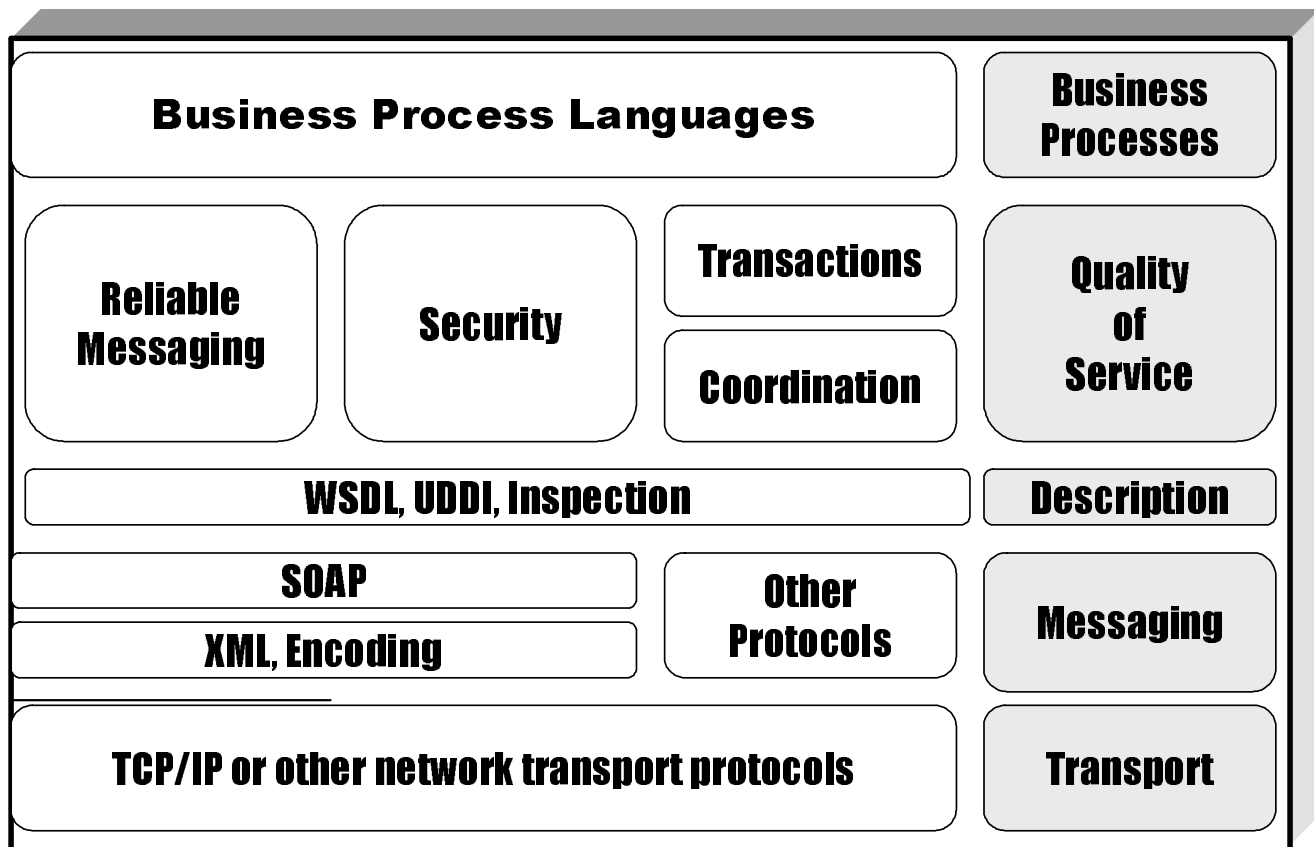


Figure 2: Web Services Stack

to the e-tailor for help (or worse). In effect, the e-tailor's enterprise has enlarged to include not just the computers in its own data center, but also the thousands of client systems in which its Web Serverlets have been embedded.

What does "reliability" entail in such an application? Setting the development and debugging process to the side, and ignoring issues associated with configuration (important issues), there are still a number of problems that may arise at runtime, and that the application itself will need to deal with in an automated manner. The following five broad cases need to be addressed:

1. A failure could cause the client system to crash while performing an operation.
2. An outage could disrupt connections from the client system to the Internet.
3. A network outage could prevent the client system from connecting to a *subset* of the data centers operated by the e-tailer (e.g. others might still be accessible).
4. Something could go wrong in the Web Service dispatcher to which the client is connected (it might crash, become overloaded, come under a DDoS attack, etc).

5. Something could go wrong on the database cluster running the back-end service.

The Web Services architecture offers different responses to these varied scenarios, depending in part on the features supported by the specific platform on which the application was built, and on the way that the system used those features. Broadly, the technology support available falls into the areas covered by the WS_Transactions and WS_Reliability specifications, and in the WS_ReliableMessaging layer implemented in support of these specifications.

WS_Transactions offers support for two kinds of transactions. Basic transactions are short-running operation sequences on one or more transactional backend services, terminated by a 2-phase commit and offering ACID properties. Business transactions script a series of basic transactions and include exception handling logic.

WS_Reliability and WS_ReliableMessaging offer support for message queuing intermediaries: services that accept requests on behalf of the client, persist them, and then issue those requests when the server becomes available (and similarly pass responses back). The request should be uniquely identified by the client, and

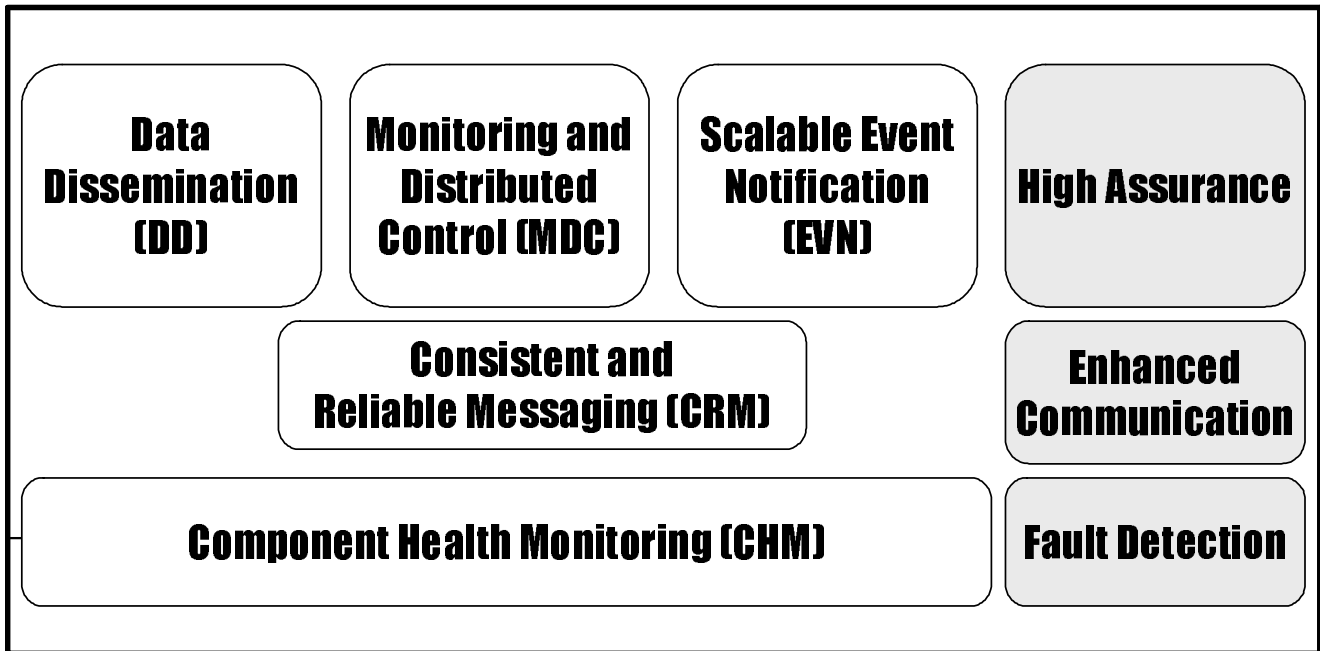


Figure 3: Components of Proposed Extensions to Web Services Architecture

the interface provides an acknowledged handoff protocol, so that the client can be sure that the intermediary has the request safely in hand. Various options permit the client to preprogram the actions to take in the event of a failure: one can specify that an operation should occur at most once, at least once, or “exactly once.” Implicit in the standard is the assumption that if the intermediary fails after acknowledging a request, the client won’t retry through some other intermediary – instead, it should wait until the failure is repaired, at which time the intermediary will take the appropriate action to push the operation forward.

If we look closely at our list of potential outages, it should be clear that these mechanisms respond to some of the issues, but not all of them. WS_Transaction guarantees that servers won’t be left in an inconsistent state if a client crashes during a multi-operation sequence, but is probably not needed if the client is performing just a single operation on a single server. WS_Reliability offers ways to issue a request while a server is inaccessible, and also to reissue a request without fear that it will be performed more than once, provided that the server itself supports the necessary mechanisms to identify duplicates and store replies. Most often, these properties would require a true message queuing intermediary that logs operations and replies to disk, a potentially costly action. Moreover, as just noted, ensuring that a request is performed exactly once in this architecture may entail waiting for a failed component to restart – a delay that could involve many minutes.

In summary, although we do have ways to handle many kinds of exceptions in the Web Services architecture, it lacks lightweight ways to guarantee high availability, where the application is trying to perform a single operation rapidly and simply wants a quick response. The architecture seems to be designed with a different style of application in mind: one in which a rather hefty back-end server is being used, in which infrequent but potentially rather long delays are basically acceptable to the designer, and in which waiting for a failed component to reboot and relaunch associated services is simply a fact of life.

If we think of Web Services as a gateway to very heavy-weight applications, these would be reasonable assumptions. Indeed, one approach holds that it is a mistake to expect prompt response from a Web Service – “Web Services are not Distributed Objects.” Revisiting our scenario with this model in mind, one might argue that the client would have been wiser to hand its request to a message queuing subsystem. That subsystem would log the request, and then pass the operation to the server. The client can then poll the response queue for a reply.

The main objection to this pipelined queuing approach is performance. We’ve taken a transaction that might have been possible with a few milliseconds of delay and introduced what will probably be many seconds of delay, if not minutes when a failure actually does occur. Moreover, the approach introduces a very large *variance* in the expected delay. To the extent that application designers try to use Web Services in light-weight object

oriented applications – applications where we might in the past have used CORBA technologies, for example, all of this may seem excessively expensive and slow. This raises a basic set of questions.

First, is the pipelined approach to reliability really necessary? What can be done to offer high availability in a Web Services environment? In what follows, we'll argue that even very limited process group replication mechanisms could go far towards supporting a high availability and rather lightweight reliability option for Web Services, by leveraging the mechanisms already described but eliminating their need to persist data to disk and to wait for failed components to restart. By getting these very costly steps out of the path, we can offer the sort of quick responses that one expects in other kinds of object oriented settings.

But a second and broader issue is also apparent. If we simply offer mechanisms capable of overcoming detectable failures, the application using those mechanisms might still suffer availability problems for other reasons. In many settings, the ability to *sense* a problem is itself a problem. As we look towards the sorts of ambitious 4-tier applications cited earlier, we should also recognize the need for mechanisms capable of helping applications monitor the state of a system, diagnose problems, and orchestrate a response when a problem occurs. Moreover, much of this functionality will be needed not just within the data centers that host servers, but also on the *client* side of a Web Services application – the little serverlets handed out by our e-tailer, for example, will need ways to monitor their health, their connectivity to the data center, and to react in a loosely coordinated way if disruptions occur. All of this leads to a set of proposed Web Services extensions, described in the remainder of this paper, that both solve the specific problem before us and also open the door to all sorts of new applications.

4. Reliability: The Real Requirements

Before plunging in, it may be helpful to review some of the background knowledge in the area of highly available, self-managed distributed applications. Our group has worked in this area for twenty years, first developing the Isis Toolkit, then a series of follow-on systems (Horus, Ensemble and JavaGroups; the latter is a component of the popular JBoss platform). Most recently, we developed two extremely scalable technologies: Astrolabe and Bimodal Multicast. Of these, Isis gained the widest acceptance – it is still used today in the New York and Swiss Stock Exchanges, the French Air Traffic Control System and the US Navy's AEGIS warship. These experiences lead to a number of insights.

1. *Consistency (in both an informal sense and a more mathematical sense) is at the core of predictable, highly available applications.* A fundamental aspect of consistent behavior concerns agreement on system membership [3, 5] If different components have different, confused, notions of which components are healthy and which are faulty or degraded, this will be reflected in a confused higher level behavior. Consistent failure detection and reporting can orchestrate consistent reaction. In much of this work, one finds that the most fundamental notion of consistency concerns tracking the components that are operational members of the system. Consistent (agreed-upon) membership information can drive all sorts of higher level mechanisms.
2. *Process group replication techniques can be used to obtain very high performance cluster-style implementations of critical services.* However, the groups need to be kept small. The techniques we understand best work well with 3 to 5 group members, but scalability problems are evident in larger settings. The Isis Toolkit provided such functionality as data and service replication and synchronization, monitoring of system status, help in launching a new program and integrating it into a running system, event notification (also called publish-subscribe), and reliable data dissemination (multicast). However, to reiterate the point, these tools work only for relatively small server groups, albeit with potentially larger numbers of clients.
3. *A solution must enable "local" interventions.* For example, in our e-tailer scenario, both the client system and many of the backend applications are likely to be legacies, hence difficult to modify. Any new mechanisms need to operate primarily within the Web Services dispatching component. Having done this, of course, there may also be an opportunity to extend the platform on the client side, and developers of new Web Services servers would presumably take advantage of all available technologies. But at least some basic set of functions need to work even if the clients and servers are unmodified.
4. *Large-scale systems need mechanisms for monitoring their own state, diagnosing problems, and initiating repair when necessary.* Here, we begin to step beyond the basic issue of high availability by shifting attention to questions of large-scale management and control of an application that may include components on thousands or even tens of thousands of sites,

scattered over the Internet and in many cases, behind firewalls and network address translators. Although one can build mechanisms that scale to this degree, it isn't easy, and few off-the-shelf technology options exist for such settings. Mechanisms that respond to these monitoring objectives would also be useful in other settings.

Two broad observations follow from these comments. First, we note that high availability solutions depend up replication of underlying critical components. Additionally, we note that high availability doesn't come about by accident. And this should be a source of concern, because Web Services, for the time being, has lacked a systematic set of mechanisms aimed at applications demanding rapid response times as opposed to pipelined on transactional forms of reliability.

5. Extending the Web Services Architecture

Our arguments lead to the proposed extensions shown in Figure 3. The first two components focus on high availability and replication:

- **Component Health Monitoring (CHM).** This module represents a new service used to track the health of individual Web Services components. The service might be imagined by analogy to the Internet's Domain Name Service: The DNS maps host names to IP addresses, while the CHM maps component identifiers to health information. To carry out this service, the CHM would watch the monitored components and report changes in their state in a consistent manner to all components "interested" in it. The data would then be used to detect failures and trigger rollover or other compensation actions. For example, a Web Service client connected to a data center would be said to "have an interest" in the health of the services running on that center.
- **Consistent and Reliable Messaging (CRM).** The CRM layer basically offers a simple process group mechanism, limited to small groups that use virtual synchrony for replication. CRM offers both a group communication interface and a second TCP interface in which one or both endpoints of a standard TCP stream can be replicated over a set of group members. In this mode, CRM offers a form of *unbreakable TCP endpoint*. The idea is that when WS_Reliability is used to talk to a group over a TCP connection that terminates in such a replicated manner, we can achieve safe handoffs without needing to persist information onto a disk, and can tolerate failures without waiting for failed components to restart. CRM can also support other styles of group applications in which data is

replicated, but is not as ambitious a group computing toolkit as we supported in our past work on Isis, Horus and Ensemble.

These two components of our system are designed to benefit even legacy clients and servers – they can be used exclusively in the router component of a Web Services platform and the result will simply look like a very reliable router that remains operational even when some of its components crash and restart. The remainder of our platform aims at a new set of clients and servers that are designed with high availability and "autonomic behavior" in mind. These include:

- **Data dissemination (DDS).** The DDS component would provide reliable multicast mechanisms for use in replicating data within critical servers and for streaming styles of broadcast from Web Service systems to their clients. Once a server can be counted upon to remain operational, we believe that it will often be important to stream data of various kinds from it to its clients. Thus DDS needs to operate at Internet scale and to be useable even in the presence of firewalls and address translators.
- **Monitoring and distributed control (MDC).** The MDC component responds to the need for mechanisms capable of monitoring and managing the entire system, by tracking performance metrics and other state variables and reporting them out. In particular, whereas the CHM service is used to detect failures of *individual* Web Services components, the MDC service looks at *aggregated properties* of the system as a whole. For example, in our e-tailer scenario, MDC might be used to detect a problem preventing large numbers of serverlets within a hospital from connecting to the e-tailor's Cleveland data center. Such a condition may not involve the failure of any part of the Web Services platform – it could arise from the Internet itself, and may be detectable only by collecting access statistics from large numbers of clients and correlating them.
- **Event notification (EVN).** When an event occurs it may have system-wide implications, and waiting for applications to notice the new situation isn't always appropriate. For example, perhaps a data center that was online is about to go offline and needs to instruct clients to roll-over to specific alternative servers. An event notification could tell them to do so. DDS and EVN play related roles, but whereas the DDS service focuses on streams of data sent by the Web Service to its clients, the EVN service focuses on urgent, small, one-time events. We're hoping to base the interface on the new WS_Eventing specification.

Reiterating the point made in Section 4, we see it as very important that all of these services provide strong, well specified properties to the application developer. Lacking rigorous semantics, applications layered over them will suffer from unpredictable and hence potentially unreliable behavior. Brevity precludes a very detailed discussion of this point, but in what follows, we touch on the major requirements we've identified for each of the services and offer some preliminary thoughts on how each could be architected.

5.1. Component Health Monitoring (CHM)

Component health monitoring is basically a failure detection service, although we favor a more generic term because failure is sometimes interpreted overly narrowly. After all, a server may not be acceptable for a given purpose if its mean response time is degraded, even if the server is still operational, and one can generalize this observation to a very broad comment that "failure" is often in the eye of the beholder.

Yet a converse observation also applies. Consider a simple system consisting of a primary server, a backup, and a set of clients, and used in a very sensitive setting. Perhaps, our server is an air-traffic control server that tracks status for sectors of the sky in some region, telling controllers whether it is safe to route a plane into that sector. It is easy to see that if the clients are left to make their own failure detection decisions, a "split brain" condition could arise in which some clients roll to the backup while others remain connected to the primary [3]. In this state, inconsistent advice could be given out, compromising flight safety.

One solves such a problem by introducing a system-level protocol to enforce agreement on failure detections: if a failure is sensed (by any client), a protocol runs and then *all* components monitoring the failed component are informed as simultaneously as possible about the event. The property we are after is one that is formally called the Consensus property, and one typically uses a group membership protocol to enforce it [2]. On the other hand, weak notions of failure (e.g. "service A is too sluggish, so I'll try service B") would typically not require the consensus property.

This leads to a CHM architecture supporting two levels of monitoring, one guaranteeing just a weak form of consistency, and the other offering consensus. The service itself would probably be deployed and used in a manner similar to the Internet DNS. When a component is determined to have failed, either by a representative of the service or by a client, this would trigger the appropriate protocol among the group of service representatives with an interest in the component in question, after which each representative would notify

the local components that have registered such an interest.

5.2. Consistent, Reliable Messaging (CRM)

As noted in Section 4, our team has considerable experience using process group replication techniques to build highly available applications. CRM is basically a simple, highly optimized group communication layer that supports the virtual synchrony model and can be used to replicate data or to perform a simple task fault-tolerantly. CRM lacks many of the features found in previous generations of group communication tools. We intend it as a simple, extremely fast, very lightweight mechanism with limited functionality offered to the user.

This said, CRM does offer one rather unusual group-based communication option. The Web Services architecture inherits a strong dependency on the TCP protocol as used by Web Browsers. Most clients will use TCP to talk to Web Services. However, TCP is a two-party protocol, and this causes problems: Web Services interpose at least one process between a client and the server it is contacting, giving rise to the many scenarios cited in Section 2. WS_Reliability handles this by having the intermediary take responsibility for the request by making it persistent, but if that intermediary fails, the connection breaks and the client may need to wait for the failed component to recover. Replication of the intermediary processes is clearly needed for high availability.

Accordingly, we are designing an extension to TCP using techniques developed in our work on process group replication, but that never entered into wide use. Basically, these allow us to connect a *normal* (unmodified) TCP client to a *group* of processes that jointly manage a server-side TCP endpoint. The resulting shared endpoint allows the set of servers to cover for one-another.

It is natural to wonder how costly this form of replication will be. To accomplish it, one server within the group is elected as a coordinator (the top one, in Figure 4), and it broadcasts every incoming IP packet associated with the TCP connection, permitting group members to maintain identical endpoint state [2]. Indeed, even timer events are multicast, so that every action that can change the state of the server-side TCP endpoint will be seen in the same order by all replicas. Should the coordinator fail, this means that one of the replicas can take over by rebinding the IP address associated with the endpoint and then resuming action just as if it had been the coordinator all along.

In our work on the Isis, Ensemble and Horus systems, we found that highly optimized group replication protocols for settings such as this can run at a rate of

80,000 or more events per second – and this was with technology that is now several generations old. Accordingly, we believe that replication can be cheap enough to pass largely unnoticed, particularly given the many other overheads in the Web Services architecture.

The “endpoint” group can then use the WS_Reliability acknowledgement protocol to interact with its client, functioning as a “high availability intermediary”. This approach permits us to achieve extremely high levels of availability without sacrificing performance in the manner seen when using message queuing intermediaries. Note that the CRM module could also be used for other kinds of replication in server or even client applications, an option we believe will open the door to building new kinds of high availability servers.

5.3 Data Dissemination (DDS)

CHM and CRM are of potential value to legacy clients and servers, because they can be used transparently by a set of Web Services routers without changing the client or server applications. However, we now describe a series of platform features, aimed at situations in which the client side will play an *active* role in self-management. The DDS module provides reliable multicast-style data streaming from the Web Services platform to a potentially large number of clients that must link directly to the DDS protocol. Our DDS framework standardizes such notions as joining a group, sending a message, and delivering a message, but offers plug-in flexibility with respect to the actual properties of the protocol. The current thinking is to exploit the approach we used in our Horus and Ensemble multicast systems, both of which permit the user to “snap in” a protocol stack consisting of one or more multicast microprotocols, each concerned with a specific property. For example, one microprotocol could offer data encryption, while another is concerned with hiding out-of-order delivery and yet another with the virtual synchrony reliability property.

A given application would assemble a stack of protocols having the desired composite property and snap it into place, then would use the standard API to send and receive messages. We anticipate that once a group has been created, its properties would not be changed on the fly, although there is prior work on that problem and this decision could be revisited in the future.

5.4. Monitoring, Distributed Control (MDC)

As noted earlier, we believe that monitoring a system “as a whole” poses distinctly different challenges than monitoring its individual components. Our group developed *Astrolabe* as a response to these needs [4].

Astrolabe works by monitoring the dynamically changing state of a collection of distributed resources, reporting high quality “local” data and summaries of remote information collected system-wide to its users. It organizes this data into a hierarchy of domains, which we call zones, and structures each zone as a small relational database – a table, with a row for each underlying zone or system component, and a column for each of a set of monitored attributes. Attributes may be redefined while the system is running, and updates propagate within seconds, even in huge networks. A novel peer-to-peer protocol is used to implement the *Astrolabe* system, which operates without any central servers.

Much of the power of *Astrolabe* stems from its ability to support online data mining and data fusion. The system continuously computes summaries of the raw data it monitors, using on-the-fly aggregation. The aggregation mechanism is controlled by SQL queries, and operates by extracting summaries of data from each zone, then assembling these into higher-level database relations. By reprogramming these features on the fly (a task very much like asking a database to compute a dynamically materialized relation), a human user can reconfigure *Astrolabe* within seconds, causing the system to begin tracking information that it may not even have been instrumenting before the request. Thus, as the needs of its users change, the behavior of the system can adapt to respond to those new requirements. (Aggregation can also be valuable even if the “user” is actually the application itself, but in this case the aggregation queries would be predefined ones and wouldn’t change while the system is running). The speed and agility of the technology open the door to a completely new way of viewing the system monitoring and control task.

Astrolabe’s aggregation mechanisms are analogous to database queries. When the underlying information changes, *Astrolabe* will automatically and rapidly recompute the associated aggregates and report the changes to applications that have registered their interest. Even in huge networks, any change is soon visible everywhere. For example, suppose that a few servers in a data center come under a distributed denial of service attack. Suspecting this, an administrator might ask *Astrolabe* to capture some sort of statistic symptomatic of attack – perhaps, the rate of incomplete attempted connections to each server. In doing so, *Astrolabe* might also be asked to begin collecting instrumentation of a type that it had not previously been monitoring.

Astrolabe has potential access to a great variety of host-maintained statistics and can also tap into data maintained by the application or even stored in files and databases. Thus, subject to user-enforced permissions, a wealth of information is potentially available to the

individual operating the system, as well as to application programs that exploit Astrolabe as a standard infrastructure for capturing system status data. We see Astrolabe as a new kind of system-wide service that, if deployed widely enough, could encourage a new generation of applications that adapt under stress more rapidly and more automatically than is possible in the absence of such services. The value of Astrolabe in such a setting is multiple: It brings standards to the monitoring task, so that all aspects of an application fall under a single umbrella. It offers a form of “one-stop-shopping”, bringing standardization to the way that monitored data is delivered to users, both human and programs. And it offers robustness and security, which are often lacking when such problems are tackled in ad-hoc ways.

In a Web Services application, we believe that Astrolabe can be used to create new client-side options for detecting and responding to problems such as difficulty accessing a data center, while also helping the administrator of the data center manage the application as a whole and diagnose failures that might require intervention. The Astrolabe epidemic protocols often route around network disruptions that prevent TCP connectivity, hence Astrolabe will usually be operational even under conditions where other Web Service protocols are disabled and hence not useful.

5.5. Event Notification (EVN)

The last major component of our architecture is still at an early design stage. Our current thinking is to support a form of distributed query processing, in which the components of a Web Services system are treated as small databases containing one or more “tuples” that can be queried (very likely, the same tuples from which Astrolabe extracts its data). Whereas Astrolabe works to continuously monitor and aggregate all of this data and limits itself to a fairly small amount of data, we envision an EVN service that would start by doing more work locally: looking at a potentially large amount of data on each node, and watching for conditions of interest. When a condition arises, the system would notify applications watching for that condition. Ideally, we believe that such an approach can support true queries: the application program would express a relational query over the “system state”, and we would compile this down to local actions, then finalize the query evaluation by combining the local results within the network. Rather than speculate, however, we leave further details to some future treatment. As noted earlier, we’re hoping to base the interface on the new WS_Eventing specification.

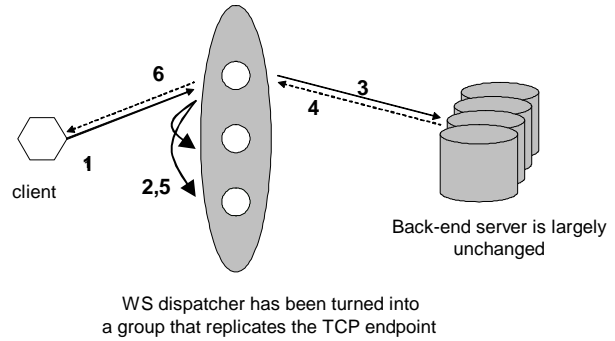


Figure 4: A reliable client-server interaction in our proposed architecture

6. Pulling it All Together

How would our architecture respond to the needs of the e-tailer cited as an example in Sections 1 and 2? In this Section, we briefly walk through the architecture. Figures 4 and 5 illustrate the approach.

The basic solution we envision starts by replicating the state of the Web Server intermediary on 2 or more nodes, offering fault-tolerance against disruptions that might occur during request processing. We use the replicated TCP functionality of the CRM component here, hence a client can use a traditional TCP implementation, and yet its actions are replicated across a set of servers. Similarly, when the backend database application interacts with the server group, information is replicated across the group members. To exploit this initial feature of our solution, no changes are required in the client or server systems.

The remaining failure concerns enumerated at the outset can be tackled using other components of our architecture, or by exploiting the mechanisms already proposed as part of the WS_Reliability and WS_ReliableMessaging standards. Tackling the self-management aspects of the problem using our tools requires some changes on the client side and hence is feasible in the scenario of Section 2, where the e-tailer developed the serverlets and was in a position to link them to our package, but might not be feasible in some other setting, for example one in which a similar set of issues arise but there is no functionality “owned” by the Web Services platform developer running on the client systems.

We anticipate using the DDS, MDC and EVN components of our platform for several purposes. Using DDS it is possible to stream information to client systems, such as continuous reports on inventory levels, pricing, or (moving away from the etailer scenario) other sorts of soft real-time data. MDC would be used by the Web Services client systems to report their status, and in

particular to share information about performance obtained from the various data centers

As illustrated in Figure 5, clients could thus sense one-another's problems and when a pattern arises ("nobody can get through to Sunnyvale"), would be in a position to undertake a coordinated, clean, roll-over to a backup center. Meanwhile, MDC would let the administrator of the data center detect problems very quickly and address them even before end-users are aware of a difficulty. One can easily identify additional MDC uses for purposes such as load-balancing, sharing of information in client-side caches, etc.

MDC also has applications within the data center itself. Today, we have relatively few monitoring options for large, complex applications spread over many machines (companies like Google, eBay and Amazon report that their data centers have thousands of machines), and perhaps also over multiple geographic locations. Better data would permit applications to become more self-controlled: adaptive, self-configurable, and self-repairing.

EVN would be used for "ad hoc" notifications. For example, while we use DDS where there is a long-term relationship between data consumer and the server producing that data, EVN might be used to notify the clients connected to such and such a server that the server will be shut down for maintenance in 5 minutes. In systems that do extensive caching (likely to be important in high-performance Web Services applications), EVN could be used for notification when a cached data item changes or must be invalidated. More broadly, EVN can play roles similar to those of publish-subscribe systems, although (unlike most such systems), our architecture is designed to function at Internet scale and to offer a more flexible range of reliability guarantees.

9. Conclusions

A review of the Web Services architecture reveals many issues that could limit high availability, reducing the perceived reliability of the architecture for applications unable to work in a transactional paradigm and for which the pipelined style of reliability favored in the WS_RELIABILITY specification would be inappropriate. The problem may be particularly acute in interactive applications that treat Web Services as if they were distributed objects. We propose a set of extensions to the basic architecture that work within the standards, yet are able to respond to many of these needs. Our solutions offer a basic form of fault-tolerance where client systems must remain unmodified, and a more sophisticated fault-tolerance property where the client can be linked to a new library.

10. References

- [1] K. P. Birman A Review of Experiences with Reliable Multicast. *Software Practice and Experience* Vol. 29, No. 9, pp, 741-774, July 1999.
- [2] K.P. Birman. *Reliable Distributed Systems*. Springer-Verlag. 2004. (Pages 319-329 discuss the fault-tolerant TCP endpoint proposed in Section 3.2.)
- [3] B. Glade and K. Birman. Reliability Through Consistency. *IEEE Software* (Special Issue on Safety and Reliability), May 1995.
- [4] R. van Renesse, K. Birman and W. Vogels. Astrolabe: A Robust and Scalable Technology for Distributed System Monitoring, Management, and Data Mining. *ACM Transactions on Computer Systems*, May 2003, Vol.21, No. 2, pp 164-206
- [5] W. Vogels. Tracking Service Availability in Long Running Business Activities, in *Proceedings of the First International Conference on Service Oriented Computing*, Trento, Italy, December 2003.
- [6] WS_RELIABILITY, WS-ReliableMessaging, WS_Eventing. W3W Working Drafts. August 8, 2003. <http://www.w3.org>

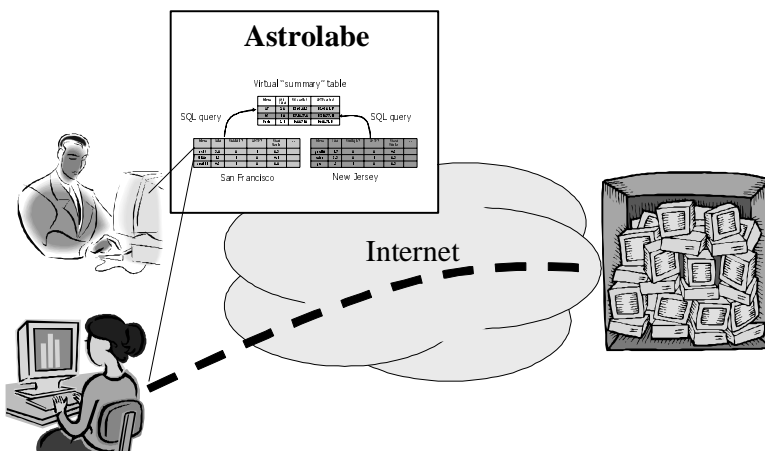


Figure 5: In a traditional Web Services architecture, clients have no information about data center status. The MDC (Astrolabe) lets clients track one-another's status and that of the center. They use it to diagnose problems and react in a coordinated way.