Comparison of UDDI Registry Replication Strategies

Chenliang Sun, Yi Lin, Bettina Kemme
School of Computer Science
McGill University
Montreal, Canada
{csun1, ylin30, kemme}@cs.mcgill.ca

Abstract

UDDI registries are intended to become the world-wide lookup mechanism for web-services. As such, the registry has to provide high throughput, low response times, high availability, and access to accurate data. Replication is often used to satisfy such requirements. Various replication strategies exist, favoring different subsets of the above performance metrics. In this paper, we have a closer look at two very different replication strategies. One strategy follows the UDDI speci£cation, the second uses a middleware based replication tool. In this paper, we provide a comparison of these two approaches focusing on performance and ease of integration with an existing UDDI implementation.

1 Introduction and Motivation

Universal Description, Discovery and Integration (UDDI) is a speci£cation for distributed web-based information registration of web services [3]. A UDDI registry stores information about service providers and their web-services. Service providers are typically companies, organizations, or institutions. The information stored in a registry follows a relatively straightforward schema, and many implementations use a relational database system as storage manager. The interface to a registry provides two main functionalities. Firstly, the information in the registry must be maintained, that is, it can be registered and updated. Secondly, users can query the registry to retrieve information about service providers and their services.

With the rapid development of web services technology, web services are becoming the standard interface for B2B and B2C interaction. For those applications, the entries in the UDDI registry are likely to be modi£ed seldomly, but the read load can become very high. However, we envision that more and more other types of applications will take advantage of the web-service paradigm. These applications might be quite different in their nature, and hence put different requirements on the functionality of UDDI registries. For instance, web-services are an attractive computing paradigm for peer-2-peer and grid computing environments, where individual sites are willing to share CPU and storage resources in order to cooperate in a common computation [18]. Finding resources, evaluating the capacity of sites to execute certain services, and deciding on work distribution are crucial tasks in these environments. UDDI, or at least deviations of UDDI, could be an important building block to support performing these tasks. They might not only store information about the services different providers offer and on which sites they are located, but also the capacity and connectivity of these sites, and maybe even their recent average load. However, such information is much more volatile, and will change frequently. Furthermore, components that query the registry to determine the machines that are able to execute their tasks, require up-to-date information in order to achieve the desired level of quality of service. Hence, in order for UDDI registries to be the mediating force between providers and users, UDDI might have to handle high loads of modi£cations, and accuracy of the data is crucial to re¤ect the dynamic behavior.

In any case, as the community using web-services grows, the UDDI registry is a crucial entry point that needs to provide high throughput, low response times, high availability, and access to accurate data. Replication is often used to satisfy such requirements. Without replication, a central registry and the network links toward this site can easily become a bottleneck, and a single point of failure. Replicating the registry on several sites, the query load can be distributed, and the UDDI service is available despite the crash of individual sites. However, replication has the challenge of replica control, i.e., guaranteeing that the replica are consistent despite updates. This means that updates require communication among the registries, and must be executed at all sites. As such, replication will only lead to increased throughput if the percentage of updates is reasonably low. Furthermore, synchronizing access to data items across the

Proceedings of the IEEE International Conference on Web Services (ICWS'04)
0-7695-2167-3/04 $ 20.00 IEEE

replicas requires advanced communication and transaction processing algorithms. Ensuring that the implementation of a replica control protocol guarantees the promised degree of data consistency is not a trivial task.

This paper takes two quite different replication strategies, and evaluates their suitability for UDDI replication. Our choices cover two important classes of replica control [15]. Using *lazy replication*, an update is £rst executed and committed at one site, and coordination takes place only after the user receives the response. This provides fast response, however, data at remote sites not always re¤ects the latest updates. In *eager replication*, the replicas coordinate before the user receives a response leading generally to higher response times. The advantage is data consistency at all times, and no lost updates in case of failures. For both strategies, many different protocols have been proposed (recently, e.g., [7, 23, 9, 24, 15, 22, 20]). This paper does not attempt to invent yet another strategy. Instead, we focus on the impact of replication on UDDI. Of the existing algorithms we analyze to representatives: the lazy approach proposed in the UDDI speci£cation [2], and an eager scheme that uses a middleware based replication tool [21]. Section 2 presents these algorithms in more detail.

Another important factor when choosing a replication solution is the implementation overhead, and how easy the replication module can be integrated into an existing system. The ideal situation arises if (1) extending an existing UDDI registry with a replication component can be performed without major changes to the existing system, (2) the replication implementation is relatively independent of the exact UDDI speci£cation in order to also work with enhanced systems, and maybe even other forms of registries, (3) different UDDI registries can cooperate in a common replicated environment even if they have different internal implementations. Section 3 gives a detail description of the replica control implementation. It also discusses whether alternative integration techniques are feasible.

Both the replication strategy (eager vs. lazy) and the implementation of the replica control algorithm, have a considerable impact on the performance. Section 4 provides a detailed analysis not only of the in¤uence of eager vs. primary, but show that implementation decisions can have a considerable impact on the performance of the system. Results cover both local and a wide area networks. Section 5 discusses related work, and Section 6 concludes the paper.

2 Replication Protocols

2.1 Lazy Replication (UDDI Speci£cation)

According to the UDDI speci£cation [3], when a user inserts a new data item at a speci£c site of the replicated registry, the user becomes the *owner* and the site becomes

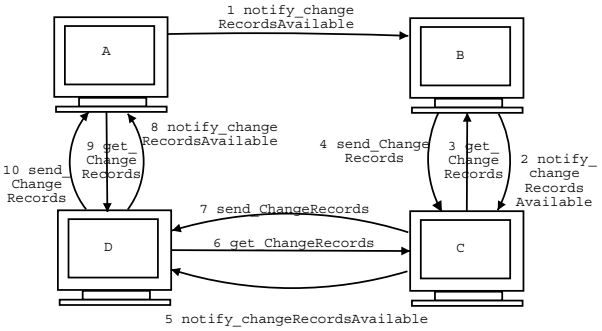

Figure 1. UDDI Replication Protocol according to Speci£cation

the primary of the data item. The primary can change if the owner wants this or because of failures. Only the owner can change the data item and has to do so at the primary. This primary generates a local sequence number (identi£er), and performs the update locally. A propagation process is started periodically propagating all update requests since the last propagation. The speci£cation suggests that all sites build a logical ring, and communication is along this ring. In case of failures backup communication paths are used. The propagation process is started at one site. This site advertises its changes to its neighboring site. If the neighbor is missing some of these changes, it requests (pulls) these changes from the advertising site, and then forwards its own advertisement along the ring. Each site keeps two additional tables. A `Status` table has a record for each site B in the system containing the latest sequence number of a request for which B is primary and A has already executed this update request. The table `ChangeRecordJournal` records all update requests. An entry describes for each incoming update request the web-service to be called, the input parameters, and which sequence number this request received.

Figure 1 depicts an example execution. Assume a UDDI registry consists of four sites (A, B, C, and D) forming a logical ring. Periodically, A noti£es B of its status table (sending a SOAP `notify_ChangeRecordsAvailable` message). B compares it's own status with A's status. If A does not have any new information, B sends its own status table to C (example of £gure). The same message exchange now happens between B and C. Assume C misses some information. It asks B to send the missing changes (a `get_ChangeRecords` request). In this message, C sends B it's own status table. For each record in the status table where B has a higher sequence number than C, B sends the corresponding records in `ChangeRecordJournal`. C executes the requests sent by B. Then, C sends its own updated status table to D. From there the process continues between D and C, and C and A. However, after one round,

Proceedings of the IEEE International Conference on Web Services (ICWS'04)
0-7695-2167-3/04 $ 20.00 IEEE

B has not yet received any changes from C and D, and C has not yet received any changes from D. Hence, A starts a second round after which all changes that existed in the system at the time A started the £rst round are guaranteed to have propagated through the entire system.

One important issue is how to determine the time interval for generating the timer event. If it is too long, the data at remote sites will be stale for a long time. But if it is too short, the communication overhead can become unacceptable, especially when the number of sites are big.

Another important issue is that all sites must be prede£ned in a replication con£guration £le. If a new site wants to join the registry, the replication site structure has to be changed, so does removing a site from a registry.

2.2 Eager Middleware Replication

Jiménez-Peris et.al. [21] propose an eager replication protocol based on group communication. The group communication system provides support for group maintenance (automatically removing failed sites from the group, joining new and recovering sites to the group), and reliable multicast. The replication protocol allows individual requests to update an arbitrary set of data items, and performs its own concurrency control to guarantee serializability across the system. Since the UDDI speci£cation indicates that only the owner of a data item can later modify it, we only present a simpli£ed protocol here. We partition the data by ownership and assume that each owner has a primary site. A client can submit an update request to any site [1]. This site will immediately multicast the request to all sites. The multicast provides a total order, i.e., although different sites might multicast messages concurrently, all sites will receive the same order of multicast messages. This order is used as execution order for con¤icting requests that want to access the same data. Since each site receives the same order, all sites will order con¤icting requests in the same way. Although the request is received by all sites, only the primary executes it, commits locally, and multicasts the physical changes triggered within the database to the other sites. The other sites then apply all changes in correct order. The approach is eager because the execution order of con¤icting requests is determined at all sites before the transaction commits at any site. If the primary fails, another site will become primary of the data the failed site owns. Hence, if a primary fails after committing but before sending the changes, the new primary will re-execute the request in the same order due to the total order multicast. Two optimizations speed up the execution [21]. (i) Since determining the total order can take a long time, especially in a WAN, the primary can start executing a request R once it receives it physically and before the total order is determined. Once the total order is determined and a con¤icting request R' should have been executed £rst (because it is before R in the total order), then R will be aborted and restarted. But if there does not exist such con¤icting request, the execution was successful and overlapped with determining the total order – reducing the overall response time. (ii) The secondary sites do not reexecute the entire request. Instead, the primary sends the physical values of the affected records. Applying such changes is much faster than re-executing the entire request.

[1] Note that this is more ¤exible than the approach of Section 2.1.

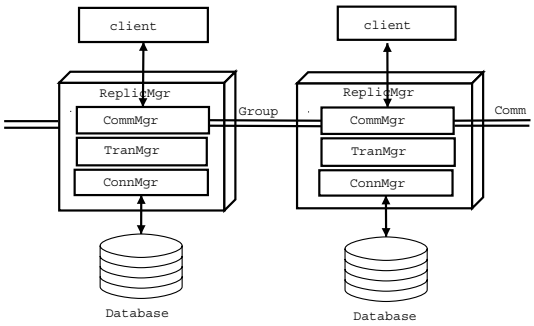

Figure 2. Middle-R Architecture

The protocol is implemented as a Java based middleware called Middle-R similar to the one presented in [21]. Figure 2 depicts the architecture. Middle-R consists of a transaction manager controlling the execution of the transactions, a communication manager that interacts with the group communication system, and a connection manager that submits the individual transactions to the underlying DBMS. We use the open-source version of the group communication system Spread (v. 3.16.2) [26]. Spread was changed to support optimization (i) mentioned above: a message is delivered to the application once when it is physically received from the network, and a second time (only con£rmation) when the total order is determined. As DBMS, we use PostgreSQL 7.2. It was modi£ed to support optimization (ii) [21]. Two functions are provided to the application, one to get the changes performed by a transaction in form of a write-set, and a second that takes this write-set as input and applies these changes without re-executing the SQL statements. The current version of Middle-R provides only a quite restrictive API. A request must be submitted in form of a sequence of SQL statements.

3 Implementations

3.1 Implementation Strategies

We can use three different approaches to extend an existing UDDI registry to support replication.
1.) A naive approach alters the existing code of all methods implementing update requests (denoted as *update methods*).

Proceedings of the IEEE International Conference on Web Services (ICWS'04)
0-7695-2167-3/04 $ 20.00 IEEE

For example, in the lazy approach, we would extend each update method in order to generate a new sequence number, include a new record into the `ChangeRecordJournal`, and update the `Status` table.

2.) In the above solution, the newly inserted code might be similar for all update methods. If this is the case, the replication related code should be put into its own class. The update methods then call the replication related methods. For lazy replication, the replication class could contain one method `record_update`, that performs the three steps mentioned above. Each update method has then a single, parameterized call to `record_update`. As such, we have concentrated replication related code into one module. However, calls to replication functionality is still scattered across all update methods.

3.) A more elegant way to weave business semantics (update methods) with the cross-cutting aspect replication is to use aspect-oriented programming. The idea is to implement the business logic as if there was no replication, and the replication module as a separate aspect. Additionally, there is a mechanism to declare that methods of the aspect (replication) should be called whenever speci£c business methods are executed. We are aware of two main ways to perform aspect-oriented programming.

- One is to use an aspect-oriented programming language like AspectJ [17]. This language is an extension of Java. It allows to implement aspects, and to declare how the aspect should be linked with the business methods. The aspect can be called before, after, or even instead of the business method. At compile time, aspect and business methods are weaved together in one executable.
- A second way is to use £lter/interceptor technology offered by current server technology. For instance, Java Servlet 2.3 [4] introduces a new component called £lter. A £lter dynamically intercepts requests and responses to Servlet to transform or use the information contained in the requests or responses. As such, an aspect can be implemented as a £lter. Filter and business logic are compiled independently. At deployment of the business logic, one has to specify which £lters should be executed before a certain servlet is called. The real "weaving" takes place only at runtime.

3.2 Original System

We use an open source UDDI implementation UDDIe [25] as our experiment platform. Figure 3.a depicts the original structure. Each UDDI site is running in Tomcat version 4.1.27. Interaction with the client (request/response) uses SOAP via HTTP. A single servlet is the entry point and dispatcher of the system. For each method in the UDDI interface there exists a Java class implementing this method. Upon receiving a client's request, a servlet parses the SOAP message, and then invokes the method of the appropriate Java class. All UDDI data is stored in PostgreSQL 7.2. The UDDI server interacts with the database via JDBC.

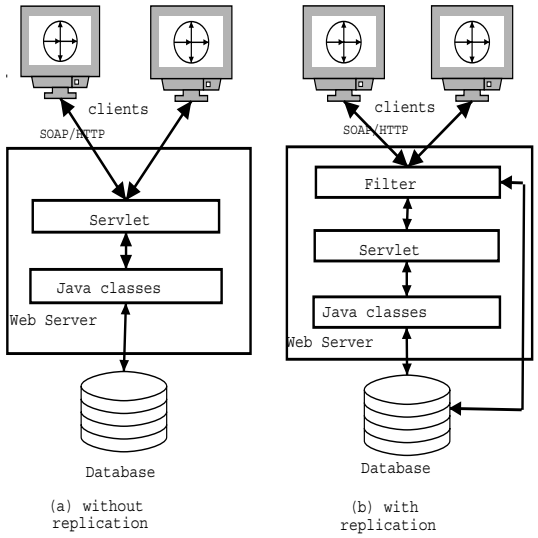

Figure 3. UDDIe Architecture

3.3 Lazy Replication

Our lazy approach uses aspect-oriented programming based on £lter technology. The new UDDI architecture is depicted in Figure 3.b. Each client request passes through the replication £lter. If it is not an update request, the £lter does nothing and immediately forwards the request to the UDDI servlet. If it is an update request, the £lter queries the message, generates the sequence number, updates the site's `status` table, and inserts the update information into the `ChangeRecordJournal` table. These operations occur within the context of a single database transaction called "logger". Then the £lter forwards the request to servlet. After the update method £nishes, the response again passes through the £lter. If the method was successful, the response is sent back to the client. Otherwise, the logger transaction will be compensated to undo its effects. Although the logger and the update method run in different transactions, the net effect of both transaction is as if everything had executed in a single transaction. The response time of the client, is the sum of both transactions. The system uses a connection pool to the database for optimization.

Update propagation is independent of the normal request processing since it completely relies on the information in the `status` and `ChangeRecordJournal` tables. It follows exactly the speci£cation, using SOAP messages to communicate between the UDDI sites. The original servlet was extended to be able to receive the new SOAP message

Proceedings of the IEEE International Conference on Web Services (ICWS'04)
0-7695-2167-3/04 $ 20.00 IEEE

types. Remember that the records of `ChangeRecordJournal` contain update requests. Each update request is logged in form of the SOAP message containing this update, i.e., in the same form the primary received the request. Hence, when a site receives such a record during update propagation it simply calls the same Java class that was called by the primary when it received the request from the user. The main characteristics of the implementation are:

- The original UDDIe code remained unchanged.
- The replication module is relatively independent of the UDDI registry implementation as long as the registry uses a web-server that supports £lter technology. Minor changes have to be performed to link a web-service request received through update propagation to the method within the server that executes this request.
- We believe that the implementation can also support other web-based applications (not only UDDI) with only minor restrictions. The £lter must, in general, only be able to distinguish whether the incoming request is read-only or an update request. The actions of the business logic upon an update request are independent of the replication module. This is true because at the secondary sites, the entire request will be executed by the same business method that executed it at the primary.
- The £lter introduces an additional indirection during execution for both read-only and update requests. Update requests have additional database access in form of a logger transaction. We could have implemented the logging and the standard update requests within one transaction. In this case, however, we would have had to intertwine the original UDDIe implementation with the replication component much more severely.

3.4 Eager Middleware Replication

For eager replication, the UDDI server becomes a client of Middle-R (see Figure 2). Each site has an instance of the extended UDDI server and Middle-R running. Only Middle-R connects to the database system.

Using an existing middleware tool we had to adjust to the interface provided by this middleware. As such, we had to adjust the business logic in the UDDI server. Instead of using JDBC, the SQL statements had to be submitted to the Middle-R. For some update methods, two requests had to be sent to Middle-R. That is, the implementation follows implementation strategy (2) described in Section 3.1. We can summarize the integration effort as follows:

- The original UDDIe implementation had to be changed at several places to adjust to the new interface. Replication is not transparent.
- The implementation overhead was still smaller than in the lazy approach since we relied on an existing replication tool.
- If Middle-R had provided a different interface, integration would have been different. (1) If it provided a JDBC interface, integration would have been basically for free. (2) Even an interface in which application programs can be deployed in the middleware (as servlets are deployed in the web-server) would have made the integration process more transparent. In this case, we would have kept the servlet in the web-server, and deployed the Java classes implementing the business logic in the middleware. The servlet then, instead of calling the Java classes directly, would have called the Middle-R to execute them. That is, the business logic itself would have remained the same but deployed at a different place, the servlet would have needed adjustments.
- In its current form, we do not see a possibility to link the Middle-R in the form of an aspect with the UDDI server. But we believe that a simpli£ed version of the replication protocol provided in Middle-R (as needed by UDDI) can quite easily be implemented as an aspect.

4 Experiment Results and Discussion

4.1 Parameters of the Experiments

We have run experiments in both a LAN and across the Internet (WAN). Four machines in Canada (Intel P4, 3.0GHz, 1 GB memory, Red Hat Linux) connected by a Fast Ethernet are used for LAN experiments. WAN experiments were conducted in Planetlab [5], an open, globally distributed computing infrastructure. The machines we used are located in North America (2), Asia(1) and Europe(1). They all have similar parameters (mostly Intel P4, 2.4GHz, 1 GB memory, Red Hat Linux). We did not have exclusive access to them.

Our experiments focus on response and execution times. Hence, one client is connected to the UDDI registry submitting requests serially. The requests call the `save_business` method. This method reads three attributes of one table (to verify the user authorization), and then performs modi£cations on four further tables to insert (or delete, update) business details, descriptions, discovery Urls, and contacts. . In lazy replication, we couple requests with propagation. There is only one request per propagation period. That is, our analysis of the propagation period shows the best case scenario where only very little information is exchanged between the sites, i.e., it basically captures the minimum communication and execution overhead. Each test run contains as many requests as are necessary to achieve a 95% con£dence interval for the mean that does not vary more than 3% from the shown mean.

Proceedings of the IEEE International Conference on Web Services (ICWS'04)
0-7695-2167-3/04 $ 20.00 IEEE

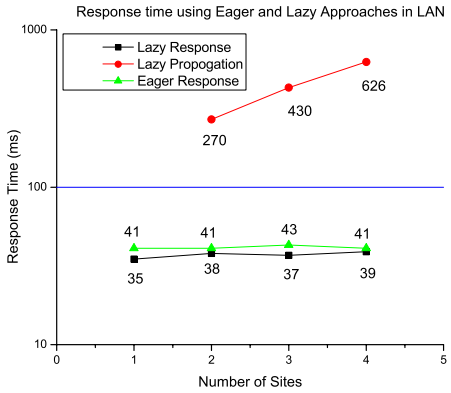

Figure 4. LAN: Response & Propagation Time

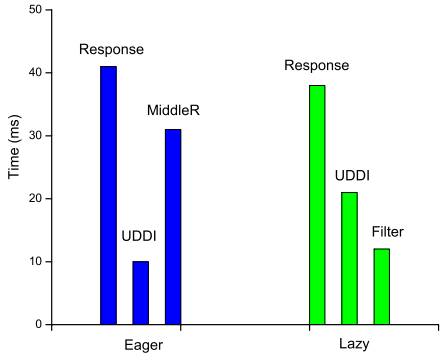

Figure 5. LAN: Execution Time

4.2 LAN

Figure 4 shows the response times for a LAN when the number of sites increases from 1 to 4 sites, plus the propagation time in case of lazy replication. For both eager and lazy replication, response time remains the same for increasing number of sites, and eager replication is only slightly worse than lazy replication (both around 40 ms).

Figure 5 provides for a 2-site system a more detailed analysis of the response time. For eager replication, the £gure shows the client response time, the time spent in the UDDI server, and the time spent in the Middle-R. For lazy replication, the £gure shows the total client response time, the time spent in the UDDI server executing the business logic, and the time spent in the £lter executing the logger transaction (both including access to the database). One can see that the eager approach spends most time in Middle-R. This includes two calls to Middle-R, the multicasts within Middle-R, the database access and the housekeeping within Middle-R. The impact of the total order multicast is not signi£cant since determining the total order in a LAN is faster than executing the request at the database. This is true even for 4 sites (and probably up to more than 20 sites). Lazy has database access overhead in the UDDI and the £lter. We consider the difference between both approaches not signif-cant, and some programming optimizations could probably decrease the response time of any of the two approaches even further. In particular, we believe that eager could outperform lazy if it were implemented as a £lter based approach instead of accessing a completely separated middleware server via RMI. If we compare the extra overhead of both approaches, then we can expect the additional database access for logging of lazy to be more expensive than the single LAN message overhead of eager.

Lazy replication has an additional performance indicator, that is the propagation time (also depicted in Figure 4). Note that eager does not need this extra effort, the data is always accurate. We can see that propagation takes a lot of time, and increases with the number of sites (note that the scale of the £gure is logarithmic). While the implicit propagation in eager replication takes a few milliseconds, it takes several hundreds of milliseconds for lazy propagation. The reason is the quite ineff£cient propagation technique, which is simple and elegant, but requires a lot of communication and processing cost. Figure 6 splits up the execution time for propagation between 2 sites. For site 2 to receive the single update performed at site 1, three messages are exchanged and several database accesses have to be submitted. Although after step 7 in the £gure, site 2 has accurate data, the cycle is repeated (what is unnecessary for two sites but becomes necessary for 3 or more sites). If a propagation process propagates more than one request then steps 1-3, and 4 remain the same. However steps 5 and 7 will take more time, and step 6 will transfer a larger message. Note also that secondary sites have to reexecute the entire web-service (step 7) while the proposed eager approach has no overhead at the web-servers of the secondary sites, and a reduced overhead at the database to apply changes. Hence, independently of the propagation interval, lazy imposes more CPU overhead than the eager approach, since the eager approach is implemented at a lower level.

As a summary, in a LAN, lazy provides only slightly faster response time than eager, however, propagation puts a considerable burden on the system. If throughput and update rates are high, propagation can easily become the bottleneck at web-server and DBMS. [21] shows that a similar middleware to Middle-R can handle considerable update rates, leaving the web-server completely unaffected.

Proceedings of the IEEE International Conference on Web Services (ICWS'04)
0-7695-2167-3/04 $ 20.00 IEEE

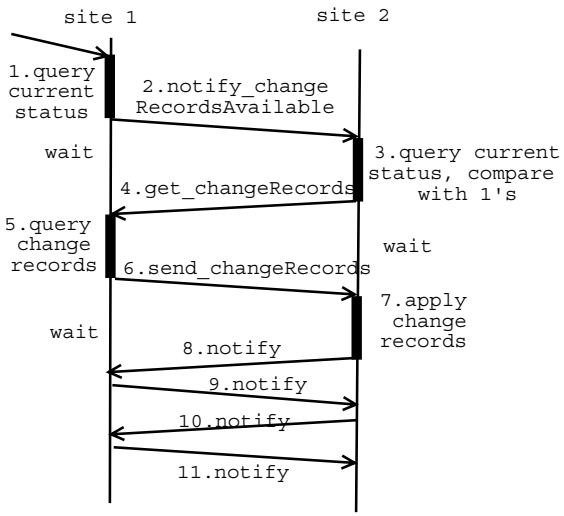

Figure 6. Lazy Propagation

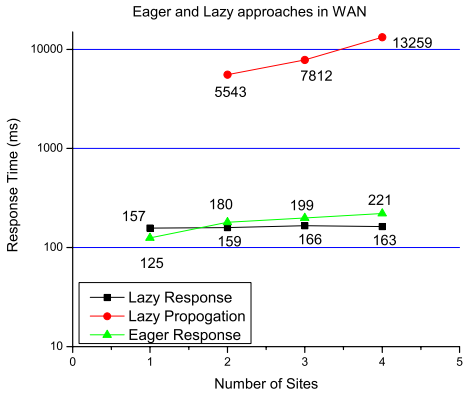

Figure 7. WAN: Response & Propagation Time

4.3 WAN

Figure 7 depicts the response times in a WAN for up to four sites. Note that this experiment uses other machines than the LAN. The machines are slower and were quite heavily used by other processes. This results in generally higher response times (three times as high for a single machine compared to the LAN experiment). There might be also higher variations in the results due to concurrent processes on the machines. We can observe that the response times for lazy remain constant with increasing number of sites since they never include communication overhead. For eager, response times increase with the number of sites. Determining the total order in a WAN takes longer than executing the updates locally. With four sites, eager has around 30% worse response times than lazy. However, the absolute numbers are still quite acceptable considering the guarantee of data consistency. Of course, the response times will further increase when new sites are added. For lazy replication, the propagation time takes now in the order of several seconds (compared to less than a second in the LAN), showing that scalability will be limited if interval time is chosen relatively small.

5 Related Work

Replication is a well-studied £eld. In regard to database replication, many eager replication strategies were proposed in the 80's [8] but never implemented because of ef£ciency problems. Commercial systems used lazy schemes instead [14]. A provocative paper of Gray et al [15] claiming that eager replication will never scale, fueled new research, both for lazy approaches ([11, 9]) providing some degree of data consistency as well as eager approaches [20, 16, 21, 6, 22] showing that eager replication can be fast and scale well if appropriate techniques are used. We have used the approach of [21] in our evalution. We believe that other approaches can be applied in a similar way with similar performance results. In the distributed systems community, object replication has received considerable attention, mainly for fault-tolerance [12, 19]. More recent approaches combine object replication and transactions [27, 13]. Some of the approaches [6, 27, 13] have looked at the integration with component based systems like CORBA and J2EE application servers. Web services and UDDI registries are often implemented using such application servers. In fact, the UDDIe registry used for our application uses a similar multi-tier approach.

6 Conclusions

In this paper we have analyzed two replication strategies and various implementation alternatives for UDDI replication. Looking at the implementation alternatives we conclude that an aspect-oriented approach is the most attractive mechanism to integrate replication with the business logic but relying on existing tools might make such an approach not feasible. Looking at the performance results an approach which allows to run replication and application in the same runtime environment has advantage over an approach that loosely couples the two components via RMI. Regarding the replication strategies, they perform similar in LANs, while the lazy approach is faster in WANs. However, looking at the absolute values, clients will probably also accept the slower response time of the eager approach. In regard to update propagation, the eager approach favors the lazy strategy in regard to overhead, and staleness of data.

Proceedings of the IEEE International Conference on Web Services (ICWS'04)
0-7695-2167-3/04 $ 20.00 IEEE

In our current work, we test our systems on more replica. Furthermore, we are experimenting with other types of aspect-oriented programming (comparing £lters to AspectJ), and implementing an aspect-oriented version of eager replication.

References

[1] UDDI.org, The UDDI Technical White Paper, http://www.uddi.org/whitepapers.html, Sep., 2000.

[2] UDDI.org, UDDI Version 2.03 Replication Speci£cation, UDDI Open Draft Speci£cation, http://uddi.org/pubs/Replication-V2.03-Published-20020719.pdf, July, 2002.

[3] UDDI.org, UDDI Version 3.0 Speci£cation, http://www.uddi.org/speci£cation.html

[4] Java Servlet Speci£cation Version 2.3., http://java.sun.com/products/servlet/download.html

[5] Planet-lab homepage, http://www.planet-lab.org/.

[6] C. Amza, A. L. Cox, and W. Zwaenepoel. Distributed versioning: consistent replication for scaling back-end databases of dynamic content web sites. In *Middleware*, 2003.

[7] T. A. Anderson, Y. Breitbart, H. F. Korth, and A. Wool. Replication, consistency, and practicality: Are these mutually exclusive? In *ACM SIGMOD*, 1998.

[8] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*, Addison Wesley. 1987.

[9] Y. Breitbart, R. Komondoor, R. Rastogi, S. Seshadri, and A. Silberschatz, Update propagation protocols for replicated database. In *ACM SIGMOD*, 1999.

[10] E. Cecchet, J. Marguerite, and W. Zwaenepoel: Performance and scalability of EJB applications. In *OOPSLA*, 2002.

[11] P. Chundi, D. J. Rosenkrantz, and S. S. Ravi. Deferred Updates and Data Placement in Distributed Databases. In *Int. Conf. on Data Engineering*, 1996.

[12] P. Felber, R. Guerraoui, and A. Schiper. The Implementation of a CORBA Object Group Service. In *Theory and Practice of Object Systems*, 4(2),1998.

[13] P. Felber and P. Narasimhan. Reconciling Replication and Transactions for the End-to-End Reliability of CORBA Applications. In *CoopIS/DOA/ODBASE*, 2002.

[14] , R. Goldring. A Discussion of Relational Database Replication Technology. In *InfoDB*, 8(1), 1994.

[15] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The danger of replication and a solution. In *ACM SIGMOD*, 1996.

[16] B. Kemme, and G. Alonso, Don't be lazy, be consistent: Postgres-R, a new way to implement database replication, In *Int. Conf. on Very large Databases*, 2000.

[17] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *ECOOP*, 2001.

[18] A. Mauthe and D Hutchison. Peer-to-peer computing: systems, concepts and characteristics. In *Praxis in der Informationsverarbeitung und Kommunikation (PIK), Special Issue on Peer-to-Peer, 26(03/03)*, 2003.

[19] L. E. Moser, P. M. Melliar-Smith, P. Narasimhan. A Fault Tolerance Framework for CORBA. In *Symp. on Fault-Tolerant Computing*, 1999.

[20] F. Pedone, R. Guerraoui, and A. Schiper. Exploiting Atomic Broadcast in Replicated Databases. In *Euro-Par*, 1998.

[21] R. Jiménez-Peris, M. Patiño-Martínez, B. Kemme, and G. Alonso. Improving the scalability of fault-tolerant database clusters. In *Int. Conf. on Dist. Comp. Systems*, 2002.

[22] E. Pacitti, P. Minet, and E. Simon. Fast algorithms for maintaining replica consistency in lazy master replicated databases. In *Int. Conf. on Very large Databases*, 1999.

[23] P. Chundi, D.J. Rosenkrantz, and S. S. Ravi. Deferred updates and data placement in distributed databases. In *In Proc. of the Int. Conf. on Data Engineering*, 1996.

[24] I. Stanoi, D. Agrawal, and A. El Abbadi. Using broadcast primitives in replicated databases. In *Int. Conf. on Distributed Computing Systems*, 1998.

[25] A. ShaikhAli, O.F.Rana, R. Al-ALi, and D. W. Walker. UDDIe: An extended registry for web service. In *Symposium on Applications and the Internet Workshops (SAINT Workshops)*, 2003.

[26] Spread homepage, http://www.spread.org/.

[27] W. Zhao, Louise E. Moser, and P. M. Melliar-Smith. Uni£cation of Replication and Transaction Processing in Three-Tier Architectures. In *Int. Conf. on Distributed Computing Systems*, 2002.

Proceedings of the IEEE International Conference on Web Services (ICWS'04)
0-7695-2167-3/04 $ 20.00 IEEE