Software Services and Software Maintenance

K. H. Bennett and J. Xu
Department of Computer Science
University of Durham, Durham DH1 3LE, UK
{Keith.Bennett, Jie.Xu @durham.ac.uk}

Abstract

Software services are being promoted as the next big step forward in software engineering.. Inevitably, both service vendor and service client programs will require maintenance. We present a service architecture that has been motivated by a long term vision for software as something which is used, not owned. This architecture is used to show how evolution of software can be achieved. It uses the marketplace to drive the process incrementally. We summarise a new fault-tolerant private information retrieval scheme for protecting users' privacy and ensuring service provision even in the presence of intentional/unintentional service provider faults (e.g. malicious failures). An implementation on a realistic distributed database suggests only a modest performance overhead.

1. Introduction

Shirky [1] suggests that web services are focussed on application program inter-operability (to aim to repeat the success achieved by the web in document access). Thus web services do not need to be invoked by a browser, but by other application programs. The most general form involves binding complex programs together from component parts anywhere in the world. General inter-operability has been tried before, but with partial success, for example DCOM, Corba, and RMI. With these, both the client and server have to load the system; with web services, the idea is to know nothing about the "other end" other than what can be communicated via standard protocols. So WSDL allows the description of a service so that a call for it can be assembled and invoked from elsewhere. In order to communicate data in a system independent way, XML is used. A UDDI registry allows service vendors to offer services, and clients to locate and call them using WSDL descriptions published along with service identification information.

It is inevitable that service offerings will evolve over time, both in terms of their functionality, and also for non-functional properties such as security, performance, and cost. For example, a stock enquiry service would want to add new query features as new investment and financial instruments are introduced. Similarly, such services tend to operate in competitive marketplaces, and pricing strategies may have to change very rapidly. Service users/clients will also need to evolve software both intrinsically and to exploit evolving services. In general, a service offering itself may use sub-services to fulfil its specification, so we have a natural recursive model which supports supply chains.

Our work on web services was motivated by studies of long term user visions for software. This led us to a general service architecture, guided extensively by the needs of evolution. We explain how evolution and maintenance are achieved in a market-place framework (we use the term *maintenance* for all forms of post-delivery support, and evolution specifically for perfective changes i.e. those driven by changes in requirements). Evolution addresses both functional and non-functional attributes, and for client and vendor organisations, web services raise immediate problems of security, privacy and prevention of malicious attack, when services interoperate between autonomous organisations. Until these problems are addressed and solved, the use of web services will be severely restricted because no-one will trust them.

Non-functional properties like security pose particular problems for evolution, since relevant design decisions tend to be pervasive through the system. We are exploring these issues by addressing the protection of users' privacy (and security) while assuring service availability. We describe some early results from research that is addressing these issues. A novel fault-tolerant private information retrieval (FT-PIR) scheme is presented that protects users' privacy and ensures service provision even in the presence of intentional and unintentional server faults (e.g. malicious failures). An error detection algorithm is introduced into this scheme to detect the corruption resulting from attack with a high probability, thereby

identifying incorrect answers. The analytical and experimental results show that the FT-PIR scheme tolerates malicious server failures effectively and prevents any information of users from being leaked to the attacker. This new scheme does not rely on any unproven cryptographic premise nor the availability of tamper-proof hardware. An implementation of the FT-PIR Scheme on a realistic distributed database suggests a modest level of performance overhead.

2. Evolution

2.1. User requirements

It is interesting to note that most published definitions of web services concentrate on technical issues [1]. For example, web services are defined in [2] as self-contained self-describing modular applications that can be published, located and invoked across the web. The W3C consortium has a definition which is similar but includes the use of XML protocols [http://www.w3.org/2002/ws/]. A web services client will then assemble or compose an application from a series of web services (although transactional costs may force an application partially to be localised). In contrast, a more radical and demand-led view has been provided in [3] and [4]. A project to establish a ten year vision for software was undertaken, and as part of this, a workshop was held with colleagues from a number of cognate disciplines to assess their current view of software [3]. This enabled us firstly to identify deep seated and strategic dissatisfaction with current software, and then identify corresponding long term visions for new software to overcome the problems.

Software vision
Interaction will be through natural forms
Software will meet necessary and sufficient requirements
Software will be personalised
Software will be self-adapting
Software will be fine grained
Software will operate transparently

Figure 1 Vision from user rationale

Figure 1 shows the categorisation of our analysis of user dissatisfaction in terms of strategies to rectify problems. Simply continuing with the current supply-side model of software production was seen to be a root cause of problems. Many problems could be traced to the unacceptable *cost of ownership* of software. The solution to addressing these issues was envisaged to be moving software from something which is owned, to something which is used. This is a common business based definition of a service [5]; it implies a demand-led view of software services. In general, users have no interest in owning software; they simply wish to use it to achieve the results they require.

As a pertinent example, users were very clearly dissatisfied with the constant upgrade and fixes for their software (for both commodity and bespoke systems). Often the change was irrelevant to their individual needs, suggesting that the granularity of change was wrong. In other words, the cost of upgrading had to be borne even when the changes or enhancements were not needed. This is a consequence of owning the software, and evolving the software is one example of the cost of ownership. Of course, the cost is not simply the cost of the new software; it is at least as much concerned with the cost of re-installation, training, and upgrading existing code and databases to work with the new release. With a finer grained structure, there is the potential to break down the upgrade into smaller parts, with the possibility that changes that are not needed do not have to be purchased. This led us to the idea that cost of ownership of software is a major problem for maintenance and maintainability, and software as a service looked to be a promising solution.

With the service-based view, we see the application structured as a set of sub-services of more appropriate granularity. Instead of maintenance, sub-services are replaced incrementally. Thus the latest or best version of a sub-service is bound in at the time it is needed; we do not have to wait to upgrade the whole application. After use, we can discard that binding, and form a new one next time the application is required. If sub-services have improved in the interim, we can bind to them instead. A useful analogy is with the development of cities; cities do not evolve according to some long term grand plan, but incrementally change, according to current needs, by replacement of buildings, utilities etc.

2.2. Business requirements

We saw that efficient technological solutions to describing, finding, binding and disengaging from remote services were required at the point of need. But a higher level, a more business oriented view is needed if web services are to succeed. Most software engineering techniques, including those of software maintenance and evolution, are conventional supply-

side methods, driven by technological advance. This works well for systems with rigid boundaries of concern such as embedded systems. It fails for applications where system boundaries are not fixed and are subject to constant urgent change. These applications are typically found in *emergent* organisations - "organisations in a state of continual process change, never arriving, always in transition".

data repository service will have to provide reliable storage, but users will also require to know how secure their data is, over what period can the data be stored, what happens if the service supplier fails as well as technical matters such as transactional behaviour, performance and privacy.

The ultra-late binding model with a high trajectory of change needs such terms to be agreed at the point

Factor	**Explanation**
Added value services	A vendor can sell a product (e.g. a camera) and then offer support services (e.g. printing) on a per use basis
From capital to revenue	To move software charging from a capital to a revenue base, to make it easier for customers to subscribe
Updates	To avoid all the problems of service packs, upgrades etc for maintenance changes
Piracy	To stop copying of pirated software
New sales	Planned obsolescence is a good way to keep business going.
Anti-trust	It may be a way round anti-trust legislation, by allowing certain external software components to be used.
Competition	On the one hand, it could open opportunities for small enterprises to enter a niche service market. On the other, established vendors may not wish them to do so
Market structure	Can be suitable for both commodity and bespoke software. Can be per use or long term agreement. Can be value chain.
Functionality	Sell "functionality" via the web to anyone regardless of platform, language etc.
Customer loyalty	Though brand names and trust – always use services by trusted supplier
Integration	Better integration of current systems using business process and modelling.

Figure 2 Business drivers

Examples are e-businesses or more traditional companies which continually need to reinvent themselves to gain competitive advantage It is not viable to identify a closed set of requirements; these will be forever changing and many will be tacit.

Any commercial agreement between a service user and a service supplier requires a set of terms and conditions to proceed. Examples of such terms include costs and payment, the legal and contractual agreements (including how disputes are to be resolved and matters of redress), warranties, and security. Some of this may be formalised as a service level agreement. Simply choosing a service on its functional capabilities is inadequate. For example, a

of need, which might require multi-party negotiation. Security (protection against threat) is such a key issue, so we decided to use this as our first exemplar of service-based non-functional attribute.

2.3. Business level rationale

As well as maintenance and evolution, a number of business imperatives favour the introduction of web services. In Figure 2, factors drawn from the popular press are summarised. These are yet mostly untested, and market momentum led by major vendors will be needed (we can compare the situation with software components, which offer a strong technical advantage yet lack many of the very powerful commercial

advantages below; hence software components, while technically attractive, remain only a small part of the software market).

2.4. Maintainability

We have been working in the field of software maintenance for many years, and we are therefore familiar with the aim, for new software, of making it maintainable. Once delivered, it should be kept maintainable. Thus maintainability refers to the ease with which it is possible to maintain and evolve software. How does the service model help maintainability?

Most existing work has sought technical solutions to this. Some (for example by Lehman) has stressed process aspects. Other work has identified "ilities", for example testability, comprehensibility, modularity, and so on. Largely, this has been unsuccessful, because (1) evolution is undertaken by humans, whose skills are a crucial factor (2) evolution causes the software to degrade (3) no adequate metrics currently exist (4) the hard changes are those of which the original designers cannot even conceive. A common rule of thumb is that maintenance changes then tend to be proportional to the size of the complete system, not the change itself.

In the service-based model, recall that at the point of need, the best/cheapest/fastest/most recent service is bound in and executed. This then rewards those services which meet the needs of the market best, and they will generate revenue; in contrast, those that fail to achieve usage will be punished by the market. It can be seen that this offers an extreme view of late binding; where services are dynamically composed at the instant of need and then disengaged afterwards. Of course this raises the question of a service request for which there is no offering in the marketplace. Although in the long term there may be technological help for automatic composition (e.g. using reflection), currently we see this as a *market failure*; where the market has been unable to provide the needs of a purchaser.

In this view, maintainability is not seen primarily as a technical problem, but of a marketplace solution. Services which adapt and change to become those that other users require will succeed. It may benefit service clients and service providers to enter longer term agreements, to adopt current standards and to offer warranties. The ultra-late binding model does not imply simply an open market in which suppliers and purchasers meet transitorily. Of course, individual services are still made of software that must evolve and be maintained, and conventional technical solutions will be employed to accomplish change. This becomes a supply side issue for vendors, and it is in their interest to ensure change is achieved as efficiently and smoothly as possible. Some evolutionary changes will remain hard; in particular simple changes which affect design decisions right across the system. An example is security.

To summarise, our vision is that software as perceived by the user will have no installation, no upgrades, no downtime, no unwanted features, no vendor lock-in, and no maintenance.

3. Web services architecture

In this architecture we have three major groups of service providers:

- **Information service providers (ISPs)**: those that provide information to other services e.g. catalogue and ontology services.
- **Contractor service providers (CSPs)**: those that have the ability to negotiate and assemble the necessary components/services to deliver a service to the end-user.
- **Software service providers (SSPs)**: those software vendors that provide either the operational software components/services themselves, or descriptions of the components required and how they should be assembled.

SSPs register services in an electronic service marketplace, using ISPs. A *service* is a named entity providing either (a) operational functionality, in which case its vendor is *a Component Provider*, or (b) a *composition template*, in which case its vendor is a *Solution Provider* (see detailed explanation below). A service consumer/client, which may be the *end user* or another service, will specify a desired service functionality. A Contractor Service Provider (CSP), which acts as a broker to represent the client interests in the marketplace, will then search the marketplace for a suitable service through a discovery process involving ISPs. Assuming such a service exists (i.e. a match can be made), the service interface is passed to the CSP, which is responsible (again on the fly) for satisfying the user needs with the service found. This will either involve interpreting this service's composition template and recursively searching for the sub-services specified there, or using the atomic service that actually delivers a result.

The CSP/broker will discover and use the most appropriate sub-services that meet the composition criteria at the *time of need*. This may involve *negotiation* of non-functional attributes with candidate services. Note that the service composition (the design activity) is *not* undertaken by the client or user, but the

templates are supplied by SSPs in the marketplace. Service providers may themselves use sub-services, so the model is inherently recursive.

It is important to distinguish binding and service composition. The *design* of a composition is a highly skilled task that is not yet amenable to automation, and there is no attempt at on-the-fly production of designs in our model. However, we can foresee the use of variants or design patterns in the future. We call this design a *composition template.* We can populate the composition template with services from the marketplace that will fulfill the composition. Our architecture offers the possibility of locating and binding such services as the service is executed. There is no concept of producing an entire executable for an application; instead, the application is constructed on the fly at the time of need from sub-services.

Prototype implementations are described in [7]. Our model has been termed a "pay per use" approach, because the user might be charged each time a service is bound as used. This is only one possibility. For example, a client organization may use a preferred supplier and pay on an agreed monthly or annual fixed cost. This can be much more attractive, since revenue costs are predictable. Also, preferred suppliers may have production processes that have been approved by the client, so issues of quality of service do not need to be negotiated on every use. This illustrates very clearly why the terms and conditions are so central to a web services approach [6], and why there is much to be done in developing web protocols such as WSDL to address such concerns.

4. Security and web services

Many security-critical information systems are now becoming accessible via the Internet. Examples of online provision include on-line census information, real-time stock information, and health data (e.g. see [4]). The information in such systems is usually stored in a number of backend databases. From a user's viewpoint, there are at least two fundamental requirements: privacy protection and a desirable level of service availability. While attempting to meet these requirements, a system has to cope with software bugs, operator mistakes, and malicious attacks ? the common causes of service interruption.

Protecting the privacy of a user is concerned with a method for protecting *the identity of the information* the user is interested in (i.e. the *intention*) against any attacks occurring during communications and on the information system side. For example, when querying an online stock information system, an investor is usually reluctant to reveal the specific stock of interest to any other parties including the operators who manage the system. This problem is called *Private Information Retrieval (PIR).* There are at least two basic requirements of such a system: high-degree privacy protection (particularly on the user side); and high availability of services. Malicious faults (resulting from malicious attacks), software bugs, and network unavailability, are common causes of problems. Existing approaches generally address the problems separately. There is little existing work that suggests a coherent solution to them.

These information sources are excellent potential applications for web services; instead of providing them simply as data sources to be queried, they can be provided as distributed services which can be invoked remotely by application programs, thereby enhancing their functionality and versatility.

A private information retrieval (PIR) scheme addresses the problem of enabling a user to retrieve data from a backend database without exposing the user's intention to the databases. This seemed to us to be a good initial problem through which to consider non-functional negotiated attributes in service-based architectures with autonomous distributed databases. It is also of major practical importance. Applications include access to financial databases, design of new drugs by pharmaceutical companies, and storage of very long lived highly confidential medical data. The first PIR scheme was introduced by Chor et al [11] in 1995, and since then it has become the subject of a significant amount of work. Existing PIR schemes and their variations assume that the databases always deliver correct answers to users' queries. There is no tolerance to the loss of answers, and no capability to detect or tolerate any incorrect answers. However, in reality, situations such as faulty database servers, malicious administrators (who may tamper with and maliciously interfere with data) and/or network partitions (resulting in some servers being unavailable or stopped) are very common. Although the replication of databases is a common way to improve service availability, it often suggests a high level of security risk for the overall system (although some replication proposals do provide certain level of protection). Once part of a system is corrupted, an attacker can gain information about the user's queries and can arbitrarily manipulate users' queries and the answers to the queries.

In this paper we present a new fault model for databases within a web services architecture, and introduce a *fault-tolerant* (or attack-tolerant) approach to private information retrieval that guarantees both users' privacy and service availability in the presence of malicious server faults. Our scheme uses *replicated databases* to achieve both fault tolerance and a controlled level of communication complexity. Our

work also demonstrates the practical feasibility of applying PIR to a realistic database system that consists of up to nine replicas, each of which keeps 46,000 records. It provides significant experimental results and performance analysis by simulating a variety of attacks on databases and on the results returned to the user. Our scheme works as follows. We assume that the database access to each database is provided through a *data access service (DAS).* By replicating databases on *k* separate nodes and limiting the communications capability of replicas, the PIR scheme can protect users' privacy when no more than *t* databases are in collusion. The user attempts to retrieve some information from *k* replicated databases, and initiates this by issuing a request to a *data access service.* Instead of sending a query to one of the replicates, the DAS sends *k* redundant sub-queries to all *k* . These sub-queries are generated by *k* query functions based on the original query. Upon receiving a sub-query, a replica will construct a sub-answer based on a predefined answer function and send the answer back to the data access service. Let *f* be the maximum number of faulty replicas. We show in [12] that $k ? (t + 1) + f$. This condition guarantees both the existence of a correct result and tolerance to up to *f* fail-stop servers. To sum up, the major contributions are as follows:

?? a new system model for fault-tolerant distributed database systems is developed, with the protection of users' privacy, applicable to service based architectures;

?? a novel fault-tolerant private information retrieval (FT-PIR) scheme is presented, which guarantees both service availability and protects users' privacy. The failure model under investigation is the most disruptive one: malicious failures.

?? the design and implementation of an FT-PIR enabled distributed database system is described. Previous PIR research has concentrated on theoretical feasibility by modelling the information stored in the servers as binary bit strings. Our work demonstrates the practical feasibility of applying both PIR scheme and FT-PIR schemes to a realistic database system and provides preliminary experimental results. The performance overhead introduced by the FT-PIR scheme is extremely modest in comparison with the PIR scheme.

The theoretical basis, assumptions and system model are given in full in [12]. Here we summarise the implementation and concentrate on recent results.

5. Implementation

We assume that all issues of SQL query interpretation are handled at the user end. The system comprises the following key parts. Each database replica has a single database, comprising (in our experiment) up to 46000 records of the same length. Each replica provides externally to its DAS a *mapping table*, which enables the record to be located by its index i given its key.

Step 1 – accept inputs: the DAS takes three inputs from the user program: an intention, ?, a schema, denoted by *s*, and a security parameter *b*. ? is typically the key of an intended record, e.g. a patient's name, and *s* is the specific field names. As *b* increases, the security of the system improves.

Step 2 – look up a mapping table: The data access service then looks up its mapping table to find the index *i* of ?. In order to provide secure and reliable services, a replica can also regularly publish the mapping table for its clients.

Step 3 – generate a random index set: we now generate a random index set *RIS* with size *b*, where the only condition is that *i* must be in the set. So *RIS* consists of *i* and other *b*-1 randomly selected indices. Since we use randomisation here, even given the same *i*, the *RIS* generated each time has a high probability to be different.

Step 4 – generate queries: using *i* and a set of random numbers *r*, a set of query functions $Q_1, \ldots, Q_k$ is used to generate queries, one for each replica 1..*k*.

Step 5 – send requests: a $request_j$ sent to $replica_j$ consists of *RIS*, *s*, and Q_j, where $j = 1, \ldots, k$.

Step 6 – generate a view: using *RIS*, the $replicate_j$ program selects records from DB_j and forms a *view*. The view is generated at runtime and will be destroyed after the operation. Two consecutive views are independent of each other. A view is a two dimensional concept, with *b* as the number of records selected.

Step 7, 8 – compute and send back an answer: based on the view and Q_j, the $replica_j$ program takes the selected data from executing the query, and uses answer function A_j to compute an $answer_j$; $answer_j$ is then sent back to the client.

Step 9, 10 – reconstruct and return a result: based on *t*+1 correct answers ($answer_{j1}, \ldots, answer_{jt+1}$), the reconstruction function *R* reconstructs the intended $Result_?$ and sends it back to the user. The function R can be executed as soon as (t+1) answers are returned. This is defined as the meaning of correct. Each result reconstructed is subject to the validation of the verification algorithm, which is summarised very briefly. Although it is difficult to develop a perfect

verification function, we are able to design a function that can identify an incorrect result with a very high probability. Full details of Q, A and R are given in [12].

6. Results

This section describes experiments to explore the effectiveness and practicability of the new implementation in the presence of various categories of *malicious* attacks, and relationship between various parameters.

We start by describing the detailed evaluation criteria of the new FT-PIR implementation. The key independent variables are: the number of replicas *k*, the actual undetected error rate *e*, and the size of a view *b*. In general, we are concerned with the measured overheads of the various stages within FT-PIR. The number of replicas indicates the number of faults the scheme can tolerate. When malicious faults occur, the verification algorithm at a DAS can detect invalid results. Since this algorithm is probabilistic, there is a small probability that the reconstructed result is actually not a correct one, which is called *an undetected error*. The percentage of undetected errors in successful reconstructions is called the undetected error rate.

Computation in the implementation is performed over a dynamic generated view rather than the entire database. The view size *b* is decided by the user and is independent of the actual size of the database. This modification has two benefits: 1) it reduces the need for high communication bandwidth and 2) it makes the security of the system adjustable.

Our experiments only simulate malicious attacks on data. Although an attacker can also target server programs or communication links, the effect on the problem we consider is the same, i.e., data is tampered with or observed by the attacker. The experiments were conducted by performing attacks on the server side. The attacks can occur before or after query processing. The first type of attack targets databases while the latter ones target views and answers. Imagine that three DAS-side threads T_1, T_2, and T_3 respectively communicate with three replicas. To simulate the attack on database, T_1 sends a request to corrupt DB_1 by randomly populating some data into the database before sending the request for processing a query. Later, $replica_1$ delivers a wrong answer. From any two answers, the main program can reconstruct a result and thus there are three possible result combinations. However, only the answers from $replica_2$ and $replica_3$ can be used to get the correct result.

The DAS machine in our laboratory is a time-sharing Sun Sparc E450 with 4 250Mhz processors running SunOS 5.8. Up to nine replica machines were used and they have the same specification as follows: 400 MHz Pentium IIs running RedHat Linux (6.0 or 7.2), 3Com EtherLink XL 10Mbit Ethernet NIC, 64 Mbytes RAM, and a 4 Gigabyte hard disk. The DAS machine resides in the campus LAN; the replica machines are connected by a 10MBit/sec Ethernet LAN which directly connects to the campus LAN. The network delay, measured using the ping command, is always less than 10ms.

The software used is: Sun J2SDK 1.4.0, MySQL 3.23, and MySQL JDBC Driver mm.mysql-2.0.4. Each replica hosts a MySQL database which can be 3000 records for some experiments, and 46,000 records for others. The use of a large database aims to demonstrate the scheme on practical size databases. The point-to-point communication channels between replicas and DAS are implemented using TCP/IP sockets. Current web service technology (such as websphere or .NET) has not been used in these experiments; our experience with them showed they would add a small systematic overhead to all results, whereas the experiments presented here show the cost of FT-PIR.

Unless otherwise specified, the standard parameter settings are: $e = 0.03$, and $b = 10$. e is set to be 0.03 because the verification algorithm is shown not to work properly when e is less than 0.03 in the current implementation [12]. b is decided mainly to save experiment time.

Although these server machines have nearly same specifications, they have been used for more than 3 years. To avoid the effect of the potential inconsistent performance of them, we make sure that each server has the same probability to be faulty by corrupting its data in turn in separate runs. For example, in the five-server case, any three answers can be used to reconstruct a result. Ten runs are carried out. In each run, the data of every two servers out of five will be corrupted.

7. Results Analysis

The experimental results presented in this paper are mainly focused on the behaviour of the system in the presence of malicious attacks, and concentrate on the performance of functions Q, A and R [12].

The graphs in figures 3 and 4 investigate relationships between the time taken to perform reconstruction and the total processing time. In fault-free situations, the reconstruction time is always below 51% of the total

processing time. However, it is clearly shown that the time taken to perform reconstruction becomes the dominant factor in the total processing time in the faulty situations. When more than three replicas were used, it is observed that 47% to 97% of the total processing time was spent on performing reconstruction process.

Figure 5 investigates the effectiveness of the verification algorithm. The dashed line is the preset undetected error rate used to perform the experiments. The solid lines are the actual undetected error rate observed. It is shown that using three replicas can deliver good actual undetected error rate since 2% to 15% less undetected errors were observed compared with the preset undetected error rate.

It is demonstrated in Figure 6 that as the preset undetected error rate increases, it takes less time to perform reconstruction in faulty situations. Respectively, 28%, 29%, 74% and 80% less time are needed to perform reconstruction when *e* is changed from 0.1 to 0.5 for three, five, seven and nine replicas.

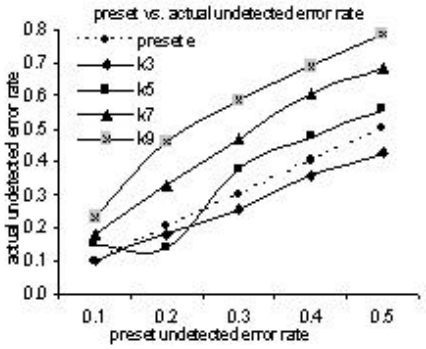

Figure 3 - the time taken to perform reconstruction & the total processing time in fault-free situations, (*b*=10, *e*=0.03, *n*=3000) vs. no. of servers

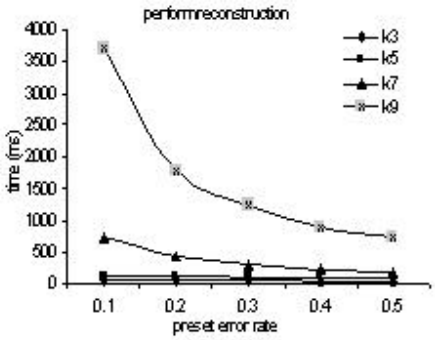

Figure 4 - the time taken to perform reconstruction & the total processing time in faulty situations (*b*=10, *e*=0.03, *n*=3000) vs. no. of servers

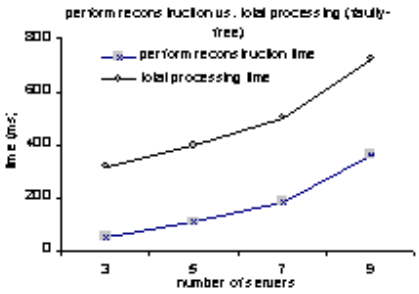

Figure 5- preset vs. actual undetected error rate in faulty situations (*b*=10, *n*=3,000)

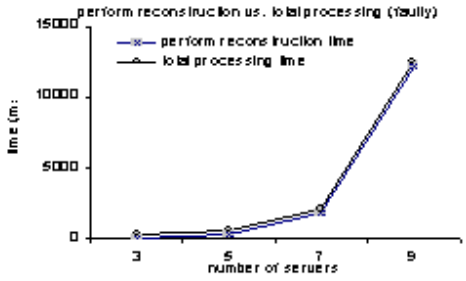

Figure 6 – preset undetected error rate vs. the time taken to perform reconstruction in faulty situations (*b*=10, *n*=3,000)

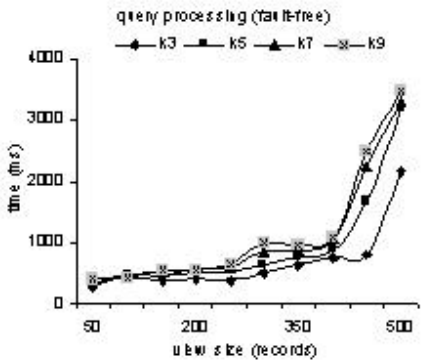

Figure 7 - the time taken to process queries in faulty situations (*e*=0.03, *n*=3,000)

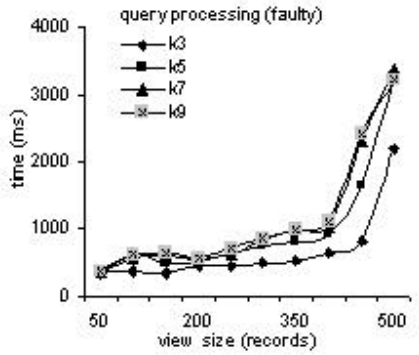

Figure 8 - the time taken to process queries in fault-free situations (*e*=0.03, *n*=3,000)

The key graphs in figure 7 and 8 show a similar pattern of query processing time in fault-free and faulty situations when the view size *b* changes. First, it is clearly that faulty status of the system does not affect the query processing process as *b* increases. On the other hand, both graphs show that the view size 400 seems to be the optimal size of views as the query processing time increases dramatically for view size greater than 400.

It is also observed that the time taken to prepare query increases as *b* increases, but the time spent on performing reconstruction is fairly stable since reconstruction is on the client side and is independent of size of *b*.

8. Conclusion

We have presented a service based architecture, motivated primarily by improving the quality of software to users in a demand-led, market based solution. This is based on ultra-late binding, when the software needed is assembled, bound, called and disengaged at the time of need. We believe that the full potential of web services will only be exploited when terms and conditions (the non-functional properties) can be incorporated fully into service models so that business drivers, as well as technical issues, can be addressed. A clear implication is that terms and conditions must also be negotiated and agreed at the point of need. We have identified a key property to be security; if this is not addressed, web services will be highly restricted in their application. Within the service model, we have explored a new approach to privacy in a personal information retrieval scenario, which is intended to fit within our late binding model.

To the best of our knowledge, our work is the first to consider the practical implementation issues of the theoretical PIR model in a real database system and also the first to provide a practical application of the FT-PIR scheme, including its application for malicious attack. It is also aimed at an ultra-late binding web service model. We have demonstrated its effectiveness against a variety of malicious attacks through significant experimental studies. It is shown that the implementation exhibits good performance.

Our work differs from recent studies in two aspects: 1) other research considers only the problem of protecting servers without considering the security issues of the client side. Our work fills in the gap by considering the problem of protecting users against malicious servers. 2) other results were not done in the presence of faults (including both fail-stop and malicious). That may be because simulating (malicious) attacks is a non-trivial task.

Acknowledgements

We wish to acknowledge the support of the UK Engineering and Physical Sciences Research Council. Much of the research has been undertaken within the stimulating environment of the Pennine Research Group (www.service-oriented.com), a group comprising software engineers from the Universities of Keele, UMIST and Durham, and we wish to thank all colleagues who have directly or indirectly influenced our research

References

1. Shirky C. *Web services and context horizons*. IEEE Computer, September 2002, page 98 – 100, vol.35, no. 9.

2. Berfield A., Crysanthis P. K., Tsamardinos I., Banerjee S., and Pollack M. E. *A scheme for integrating E-services in establishing virtual enterprises*. Proc. 12th. Int. Workshop on Research

Issues in data Engineering: Engineering e-Commerce/e-Business Systems. Pp. 134 – 142, Feb. 2002.

3. Bennett K. H., Munro M., Brereton O. P. Budgen D., Layzell P. J., Macaulay L., Griffiths D. G. & Stannet C. *The future of software*. Comm. ACM, vol.42, no. 12, Dec. 1999 , pp. 78 – 84.

4. Glover G., Bradley S., and Bennett K. H. (authors) http://www.dur.ac.uk/service.mapping/

5. C.Lovelock, S.Vandermerwe, B.Lewis, *Services Marketing*, Prentice Hall Europe, 1996, ISBN 0134558413

6. Bennett K. H.,.Layzell P. J., Budgen D., Brereton O. P., Macaulay L., Munro M., *Service-Based Software: The Future for Flexible Software*, IEEE APSEC2000, The Asia-Pacific Software Engineering Conference, 5-8 December 2000, Singapore, IEEE Computer Society Press, 2000.

7. Bennett K. H., Gold N. E., Munro M., Xu J., Layzell P. J., Budgen D., Brereton O. P. and Mehandjiev N. *Prototype Implementations of an Architectural Model for Service-Based Flexible Software*. Proc. Thirty-Fifth Hawaii International Conference on System Sciences (HICSS-35), edited by Ralph H. Sprague, Jr. p.76, 2002, Published by IEEE Computer Society, CA, ISBN 0-7695-1435-9 (Proceedings contain 316 pages with the paper abstracts only, plus a CD-ROM with the full PDF text for each paper.)

8. Bennett K. H., Munro M., Gold N. E., Layzell P. J., Budgen D. and Brereton O. P. *An Architectural Model for Service-Based Software with Ultra Rapid Evolution*. Proc. International Conf. On Software Maintenance, Florence 2001, IEEE Computer Society press, ISBN 0-7695-1189-9 pp. 292-300.

9. Bennett K. H., Munro M., Gold N. E., Xu K, Hong Z., Layzell P. J., Budgen D. and Brereton O. P. *An Architectural Model for Service-Based Flexible Software.* Proc COMPSAC 2001 (Computer Software and Applications). IEEE Computer Society Press ISBN 0-7695-1372-7, pp. 137-142.

10. E.Y. Yang, J. Xu and K.H. Bennett, *A fault-tolerant approach to secure information retrieval*, in Proc. 21st IEEE International Symposium on Reliable Distributed Systems, Osaka, Oct. 2002.

11. B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan, *Private Information Retrieval*, Proc. 36th Annual Symposium on Foundations of Computer Science (FOCS'95), Milwaukee, Wisconsin, USA, 23-25 Oct. 1995, pp. 41-51. Journal version: *J. of the ACM*, vol. 45, no. 6, 1998, pp. 965-981.

12. E.Y. Yang, J. Xu and K.H. Bennett, *Private information retrieval in the presence of malicious faults*, in Proc. 26th IEEE International Conference on Computer Software and Applications (COMPSAC2002), Oxford, Aug. 2002.