# Trading off Security in a Service Oriented Architecture

Garret Swart*, Benjamin Aziz†, Simon N. Foley‡, and John Herbert‡

*IBM Almaden Research Center, 650 Harry Road, San Jose, CA, USA
gswart@us.ibm.com
†Department of Computing, Imperial College, London SW7 2AZ, UK
baziz@doc.ic.ac.uk
‡Department of Computer Science, University College Cork, Cork, Ireland
{sfoley,herbert}@cs.ucc.ie

**Abstract.** Service oriented architectures provide a simple yet flexible model of a computing system as a graph of services making requests and providing results to each other. In this paper we define a formal model of a service oriented architecture and using it, we define metrics for performance, for availability, and for various security properties. These metrics serve as the basis for expressing the business requirements. To make trade-offs possible we also define a set of cost metrics, denominated in a uniform currency, to measure the cost of not meeting a requirement. The model, the property metrics, and the cost metrics are then used to generate a Constraint Satisfaction Problem where the objective function is set to minimize the aggregate system cost. We have written these constraints and defined realistic requirements in OPL and we have used them to generate system configurations that minimize the overall cost by optimally trading off the business requirements.

Computing systems are designed to meet the security, performance availability, and economic requirements of the procurer. Sometimes not all of these requirements are simultaneously attainable to the maximum degree. In order to get as close as possible to meeting the requirements, trade-offs must sometimes be made between the individual requirements. In order to make these trade-offs in a sensible way and to find a system configuration that best meets our overall goals, we need a model of a system which can be evaluated quickly to determine how well the system is meeting its requirements and a uniform cost model that we can use to manage the trade-offs on the different requirements.

In this paper we use formal techniques to define a precise model of a service oriented architecture, a flexible yet simple computing model that underlies the Web Service standards so popular in data processing and integration applications. We argue that this model is close enough to the reality of such systems to be interesting. We then formally define various properties of the model that correspond to important properties in real systems. The properties that we define and optimize for are different aspects of data security, server throughput, service availability and network bandwidth. We do not claim that these are the

only formal definitions of these properties that are sensible but that the ones we present are interesting and they capture important aspects of the system.

We then encode this model and properties into an Optimization Programming Language (OPL) application so that a combination of mathematical and constraint programming techniques that are part of the OPL implementation can be brought to bear on this problem to produce a set of optimal assignments of logical components to physical resources. Using the facilities of OPL, we write a system model that defines the data to be presented and its constraints. This model can be instantiated to represent any computing system that falls within the model. Once instantiated, the model can be solved by OPL to find the optimal configuration of resources that meets the requirements. This separation of the model, its instantiation and solution technique allow such systems to be used by systems administrators without a degree in operations research. Finally we show the results of this model when applied to a realistic system.

Novel aspects of this work include:

- The simultaneous modelling of important system metrics and the definition of a cost model that allows multiple business goals, defined in terms of these metrics, to be played off against each other.
- The careful definition of quantifiable security properties that correspond to properties that security experts attempt to optimize for. Security is often thought of as a binary property but the use of security metrics allows greater flexibility to the configuration process.

## 1 Modelling a Service-Oriented Architecture

In this section we define the components of our model and the information a user of the system has to provide about each component and the information that the optimizer needs to produce to specify a configuration of the system. In the next section we describe how we use this information to define properties of the system that meet the planning needs of system administrators. A UML class diagram showing the relationship between the components is shown in Figure 1.

*Service.* The fundamental system component in a service-oriented architecture is, of course, the service. We define a service to be an entity that can perform a set of operations on behalf of callers on a defined set of data. For example, Hertz may offer a car rental booking service that allows clients to book its cars. Avis may offer a distinct service that provides access to its cars. Travelocity and LastMinute may each run a travel agency that offers services that allow clients to book cars on either Hertz or Avis. These form four distinct services.

We denote the set of all services being modelled as a set named *Service*.

To specify the load generated by an invocation of a service on the server running the service we define a function *loadU* that defines the expected load units caused by a single invocation of the service.

$$loadU : Service \rightarrow \mathbb{R}^+$$

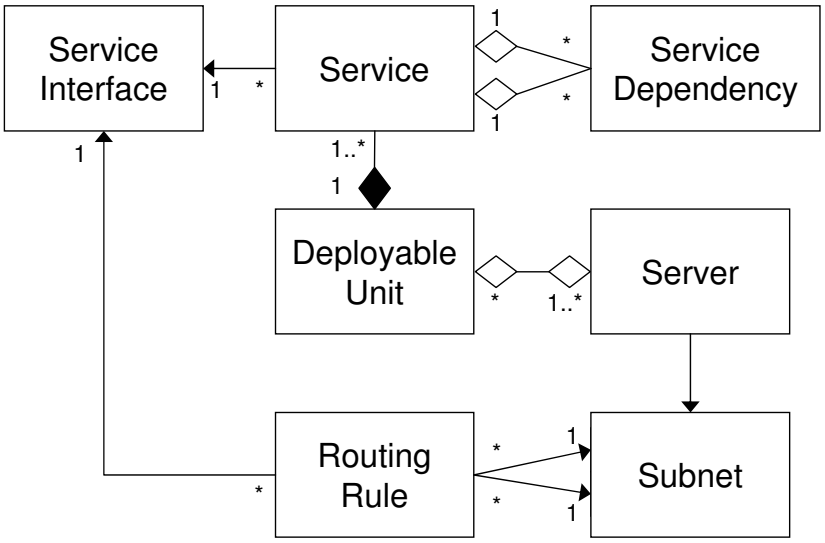


**Fig. 1.** A UML diagram of the service oriented model

*Service Interface.* Each service implements a certain protocol or language to facilitate communication with it and its invokers. We call this protocol the interface to the service. To facilitate interoperability, many services may implement the same interfaces. In a Web Services infrastructure an interface may be specified as a WSDL object and identified by a URL. In a Corba infrastructure, an interface may be specified by an IDL file and identified by a UUID.

Formally we can represent this as a set *ServiceInterface* and an *implements* function that represents the relationship between the service interface and the services that implement it.

$$implements : Service \rightarrow ServiceInterface$$

*Service dependencies.* Services may be composed from other services. For each service we assume we have a set of services that are used in this service's implementation and that we have determined the expected number of invocations of those subsidiary services for each invocation of the entry service. This can sometimes be determined by code inspection and sometimes by measurement. Even if service binding is done dynamically, data can be collected on the long-term behaviour of a particular installation.

Each subsidiary service may be used by any number of layered service implementations. For simplicity, we assume that there are no cycles in the service implementation dependency directed graph. Since this information refers only to direct dependency we call the function representing this information *dependency*1.

$$dependency1 : Service \times Service \rightarrow \mathbb{R}^+$$

We use this information to estimate the complete dependency matrix, the number of calls generated, directly or indirectly, by a single call to a service on every other service. We estimate the complete dependency matrix by computing the transitive closure of the *dependency*1 matrix. However if the complete graph has been measured, it should be used in preference to the estimate. This is the same approach used in gprof in estimating call graph values [1].

$$dependency : Service \times Service \rightarrow \mathbb{R}^+$$

*Client Service.* We define a distinguished client service whose function is to invoke the externally accessible services. The client service makes the correct mix of requests that match the expected calls from all the system's clients. The client service is special in that we do not attempt to model its internal behaviour or allocate resources to it. The distinguished client service gives us a single row of the dependency matrix to concentrate on that defines the expected call load that we are expecting for each service. Since there are no calls to the client service, one should think of the counts in the client row of the dependency matrix as representing the number of calls on the indicated services by external clients per unit time.

Formally, *client* is simply a distinguished element in *Service*.

$$client \in Service$$

In addition each service may have an availability requirement that defines the minimum probability that this service must be up and providing the needed service to the distinguished client service. We can define this requirement as a function that specifies the minimum probability that this service is allocated enough resources to perform its function. If there is no availability requirement on a particular service, the function may have value 0.

$$AvailableToClient : Service \rightarrow [0, 1]$$

Note that this function is used along with the dependency information to generate the complete service availability function in the next section.

*Deployable unit.* Each separate service is not typically deployable on a server independently. A developer or administrator will typically build or configure a set of services into a deployable unit that can be installed on one or more machines. The developer may decide services need to be collocated in the same process or on the same machine to maintain efficiency or to reduce development time. When services are combined into a deployable unit, we do not model the dependencies between these services; instead the load and dependencies are rolled up into the services that are invoked externally. The form a deployable unit takes depends on the system being used. In J2EE a deployable unit might be represented as a preconfigured WAR, or web application archive, on Linux a deployable unit might take the form of a preconfigured RPM [2] file. Unlike an unconfigured WAR or RPM file, which might contain a generic service implementation, a

deployable unit contains all information to configure the implementation to take the role as a particular service, e.g., the data it will be accessing and the other services that it may need to contact.

Formally, the deployable units are just a set *Deployable* with a function *deploys* to represent the composition of a deployable unit out of its constituent services.

$$deploys : Service \rightarrow Deployable$$

*Server.* A server is an entity on which services can be executed. Servers are not referred to directly by applications; instead applications reference services that are automatically mapped to the servers on which they are deployed. Servers are typically hardware components, though servers can be constructed logically using virtual machine technology.

For each server we have a specified failure probability. This specifies the minimum long-term probability that the server is available and providing its full execution service. This is used in the next section to compute the probability that a service is available and providing service. We specify the server availability with a function:

$$ServerAvailability : Server \rightarrow [0, 1]$$

Associated with each server is a rate at which it can perform load units, expressed in the same time units that were used for the client counts in *dependency*1 and the same load unit that was used for *loadU*. We specify the execution rate of a server with the function *powerU*:

$$powerU : Server \rightarrow \mathbb{R}^+$$

Resources on servers are assigned by the configuration system to deployable units. A deployable unit may not consume more resources on a server than it is assigned. A single deployable unit may be deployed on many different servers simultaneously, in which case the load on the component services is divided among the servers, according to the ratio of resources assigned by the server to the deployable unit. We define the number of load units per unit time allocated to a deployable unit on a server as:

$$allocU : Deployable \times Server \rightarrow \mathbb{R}^+$$

Unlike the functions defined so far, this function is not defined by the administrator, but is instead an output of the optimization process. It specifies what services a server should run and the amount of server resources that should be assigned to each deployable unit. In the next section we develop constraints that will ensure that the allocation of resources to deployable units satisfies the system requirements. The resulting allocation must not overload the server, that is the following constraint must hold:

$$\forall serv \in Server, \sum_{\forall d \in Deployable} allocU(d, serv) \leq powerU(serv)$$

*Subnet.* A subnet represents a portion of the network containing a set of servers. Servers on the same subnet can communicate more cheaply, but servers on different subnets can be protected from each other by router based filtering and firewalls. Formally, a subnet is just a set, *Subnet*, and a function *subnet* that assigns servers to subnets.

$$subnet : Server \rightarrow Subnet$$
$$clientSubnet \in Subnet$$

*Routing rule.* The filtering that can take place between subnets is represented as a set of allowable service interfaces whose messages may pass between the subnets. Typically a routing rule will be assigned to a router or firewall to ensure that only the required communication can be passed and that this required communication is safe. Like the allocation of deployable units to servers, the configuration optimization process produces the set of subnet rules.

Formally the set of filter rules is a function, *rules*, from pairs of subnets to a subset of allowable service interfaces whose messages are allowed to pass from one subnet to the other.

$$rules : Subnet \times Subnet \rightarrow \wp(ServiceInterface)$$

## 2 Properties of a Service-Oriented System

One measure of the usefulness of a model of a system is whether properties of the model can be defined that correspond to properties of the original system. In this section we present some interesting system properties that can be defined using our model and argue for their relevance.

*Service Availability Requirement.* A service's availability requirement is the probability that a service responds to a given request by one of its clients. A service's clients may include the distinguished *client* service as well as arbitrary other services that use this service. For a service to be available, in addition to the service itself being available all the service's dependencies must be available. Assuming that the availability of each request on each service is independent, we use the following constraint to define a *serviceAvailability* function that depends on the administrator-provided *availableToClient* as well as the *dependency*1 function.

$$availableToClient(s1) \leq$$
$$serviceAvailability(s1) \leq \prod_{\substack{\forall s2 \in Service: \\ dependency1(s1,s2)>0}} serviceAvailability(s2)$$

Informally this says that the service can be no more available than its constituents, but that it must be at least as available as any clients need it to be.

These constraints can be solved by starting with the services called only by the distinguished *client* service. Such a service is likely to have a nonzero

value for the *availableToClient* function. This value can be factored to determine availability requirements for each of the services it calls. This process can be repeated until service availability requirements are derived for all of the services. As might be expected this process causes lower level services to have higher availability requirements.

*Availability with Throughput.* We define execution throughput and availability constraints simultaneously, as for a service to be properly configured the probability that the service is meeting its throughput requirements must be as large as its availability requirement. An acceptable configuration must assign enough resources to each deployable unit so that with large enough probability all the services that are part of the deployable unit are getting enough execution resources to perform their function. We must also assign the resources in such a way that we never exceed the capacity of any server.

We can express the fact that a server may not be over allocated with the predicate:

$$\forall serv \in Server : \sum_{\forall d \in Deployable} allocU(d, serv) \leq powerU(serv)$$

This specifies that for all servers, that the sum of the load units allocated to each deployable is less than total load units provided by the server.

To form a predicate that insists that the needed throughput be provided with the required probability, consider a subset $S$ of the *Server* set that represents the set of servers that are available at a moment in time. For each such subset $S \in Server$ there is a well defined probability that exactly those servers are available. Assuming that the availability of each server is independent, that probability is given by:

$$setProbability(S) = \prod_{\forall serv \in S} serverAvailability(serv) \times \prod_{\forall serv \in Server - S} (1 - serverAvailability(serv))$$

That is, the probability that exactly the set of servers $S$ is available is the probability that each server in $S$ is available times the probability that each server not in $S$ is not available. For each subset $S$, there is also an expression that represents the number of load units among the servers in $S$ that are assigned to a given deployable unit, $d \in Deployable$. We compute this as a function $allocSU$:

$$allocSU(d, S) = \sum_{\forall serv \in S} allocU(d, serv)$$

For the set $S$ to have adequate capacity to be classed as being available for $d$, the number of load units allocated to the unit $d$ must be sufficient for performing the required load per unit time on the services making up $d$. We can compute

this for a unit $d$ by:

$$reqLoadU(d) = \sum_{\forall s \in Service: d=deploy(s)} dependency(client,s) \times loadU(s)$$

that is, the sum, over all services that are part of the deployable unit, of the number of invocations on that service per unit time multiplied by the number of load units consumed by each invocation. This gives us the load units required per unit time, the same units as the allocation units for server resources assigned to a deployable in *allocSU*.

The availability of a deployable unit d in a given configuration is the sum over all subsets $S$ of *Server* where the load units allocated to the deployable unit is sufficient to meet the execution requirements of the services that are part of the deployable unit, of the probability that the server configuration $S$ exists. We define the following.

$$deployAvailable(d) = \sum_{\substack{\forall S \subseteq Server: \\ allocSU(d,S) \geq reqLoadU(d)}} setProbability(S)$$

If for each deployable unit this probability is larger than the maximum service availability requirement of all the services in the deployable unit, that is, when

$$\forall d \in Deployable : deployAvailable(d) \geq \max_{\forall s: d=deploy(s)} serviceAvailability(s)$$

then the allocation of resources meets the availability and throughput requirements.

*Security Distance.* One of the important security considerations that must be taken into account when building a service infrastructure is router and firewall configuration. There are some services in which considerable skill and attention have been lavished in making sure that the service is ready to withstand the slings and arrows of outrageous hackers and other services which, while nominally secure, had best not be accessible to outside users. There are also services that store such sensitive data that best practices dictate that they should be locked away behind many levels of firewall.

One simple way of rating network service security is by the minimum number of subnet hops needed to get from the attacker to the target service. Each step along such a shortest path represents a subnet that has to be traversed and presumably hacked, in order to reach the target service. For example, in many web application infrastructures the service network is divided into 5 successively deeper subnets as illustrated in Figure 2: a content subnet, a UI subnet, a business logic subnet, a database subnet and a SAN subnet. Each deeper level provides a lower level abstraction with less fine grain access checking and often less secure authentication. Accessing each successive subnet also requires hacking a different set of systems, typically using a different set of techniques.

When configuring routing rules it is important to allow communication between subnets where it is needed, e.g. services running on those subnets need

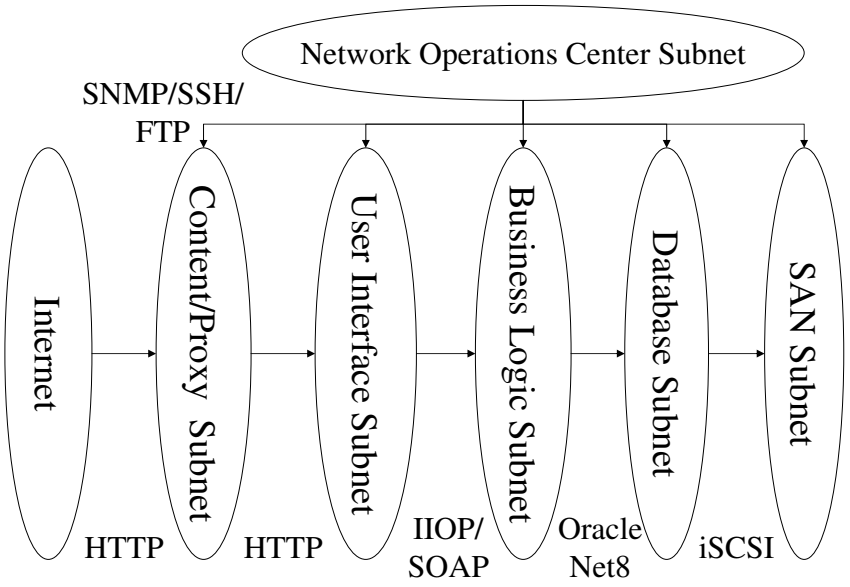


**Fig. 2.** A Typical Subnet Structure

direct communication, but at the same time we want to insist that certain services be run on servers that are deeply hidden from clients, that is there is a large security distance between the service and the attacker. Network subnet distance is a simplification of the security restrictions one might contemplate, but it is a reasonable start and it mirrors current best practices [3].

The constraint on the existence of rules allowing all needed communication can be stated as:

$$\begin{aligned}
&\forall s1, s2 \in \mathit{Service} : \mathit{dependency}1(s1, s2) > 0 \Rightarrow \\
&\quad \forall \mathit{serv}1, \mathit{serv}2 \in \mathit{Server} : \\
&\qquad (\mathit{allocU}(\mathit{deploys}(s1), \mathit{serv}1) > 0 \wedge \\
&\qquad\ \mathit{allocU}(\mathit{deploys}(s2), \mathit{serv}2) > 0) \\
&\qquad \Rightarrow \mathit{interface}(s2) \in \mathit{rules}(\mathit{subnet}(\mathit{serv}1), \mathit{subnet}(\mathit{serv}2))
\end{aligned}$$

which states that for all pairs of services that communicate, and all servers that are assigned to run those services, then the interface those services use to communicate must be present in the rules set of the router that connects the two subnets.

Given the rule above, the security distance between two services, which we will denote as $\mathit{securityDistance}(s1, s2)$, can be defined by the following recurrence.

First we define a predicate *connected* that determines whether there is a direct communications link between the two services, that is, whether any of the

servers assigned to the services are on the same subnet.

$$\begin{aligned} connected(s1, s2) = \exists serv1, serv2 \in Server : \\ allocU(deploys(s1), serv1) > 0 \wedge \\ allocU(deploys(s2), serv2) > 0 \wedge \\ subnet(serv1) = subnet(serv2) \end{aligned}$$

Then we define the security distance with the following recurrence.

$$securityDistance(s1, s2) = \begin{cases} 0, & \text{if } connected(s1, s2) \\ 1, & \text{if } dependency(s1, s2) > 0 \\ & \wedge \neg connected(s1, s2) \\ \min\limits_{\forall s3 \in Service} \left\{ \begin{array}{l} securityDistance(s1, s3) + \\ securityDistance(s3, s2) \} \end{array} \right. & \text{otherwise} \end{cases}$$

The security distance computed in this way can be used in constraints to insist that a sensitive service be a large distance from the client subnet. This can be used to restrict the optimizer from doing something silly like running a database service on the network DMZ in order to take advantage of its lightly loaded servers.

*Data Risk.* In another paper [4], a security metric based on the aggregate risk of having data from different customers make use of the same device is defined. For example, a storage service provider may decide to store data from a single commercial bank on a storage unit and to accept a level of risk $r$ in making that assignment while adding an airline's data to that storage unit may increase the risk of the assignment by a small amount but adding a competitive bank to the same unit may raise the risk considerably.

In the service-oriented context, a similar measure of data risk can be defined that quantifies the risk of placing deployable units on the same server or on the same subnet. The risk depends on the assurance level or trust we have in the server or the subnet's ability to keep the data separate and the risk associated with the information being accessed from the dependent services.

This metric is not used in the OPL implementation described in this paper, but was used in a separate OPL model described in [4].

*Network Bandwidth.* The network bandwidth used in a system can sometimes be an important consideration in system design. The internal switching inside a subnet is generally implemented by high performance switching equipment that has been optimized for network performance. Communication between subnets is performed by routers that have been optimized for security and for implementing many hundreds of complex filtering rules. Limiting the load on these expensive routers can sometimes be an important consideration.

To help express constraints or optimizer objective functions dealing with bandwidth, we define a new *traffic* function. The value $traffic(sn1, sn2, interface)$ reports the number of invocations per unit time of the given interface that may

travel between the given subnets. If the subnets are equal, the function gives the amount of intra-subnet traffic using the given interface. This function can be used to define constraints or minimize the usage of network traffic.

First we define the function *runsIn* that computes the set of subnets used for executing a given service:

$$runsIn(s) = \bigcup_{\substack{\forall serv \in Service: \\ allocU(deploys(s), serv) > 0}} subnet(serv)$$

Given this function we can define the traffic function as:

$$traffic(sn1, sn2, i) = \sum_{\substack{\forall s1, s2 \in Service: \\ i = implements(s2) \\ \wedge sn1 \in runsIn(s1) \\ \wedge sn2 \in runsIn(s2)}} dependency(client, s1) \times dependency1(s1, s2)$$

As can be seen this sums over all pairs of services where the second service implements the given interface and the services run on the given subnets. For each pair we look at the expected number of service invocations of the given type that will be requested per unit time. This is given by the expected number of invocations from the *client* to service $s1$ times the number of invocations that $s1$ makes directly to $s2$.

## 3 Optimizing a Service-Oriented System

In the previous sections we have seen how to describe a service-oriented system and how to define properties and constraints on a service-oriented system; we can now look at optimizing a service-oriented system. In mathematical programming, optimization is driven by an objective function.

The difference between an objective function and a constraint is that a constraint must hold in order to have a solution, while the objective function is merely optimized from among the solutions meeting all the constraints. While there can be many constraints in a constraint satisfaction problem, there can only be one objective function.

Some useful objective functions include those for:

- Minimizing the cost of the system. In this case the objective function may be the number of servers that have not been allocated to any deployable unit. That is to maximize:

$$| \{serv \in Server \mid \forall d \in Deployable : allocU(serv, d) = 0\} |$$

- Maximizing the security of a service. In addition to setting minimum security distance constraints, the administrator may be looking to maximize the minimum security distance from an attacker subnet to a given set of

services. That is, given a priority set of services, *Protected*, we might want to maximize

$$\min_{s \in Protected} securityDistance(clientSubnet, s)$$

- Maximizing the capacity of a system. If the load on the services may grow unexpectedly, the administrator may wish to build a system out of an existing hardware base that can respond quickly to unexpected spikes in demand by spreading any extra capacity evenly throughout the service deployments. We can compute the percentage of over capacity allocated to a service and attempt to maximize the minimum level of over capacity over all the deployable units by maximizing:

$$\min_{d \in Deployable} \left( \frac{allocSU(d, Server)}{reqLoadU(d)} \right)$$

- Minimizing the number of routing rules. Routing rules consume resources on a router and having too many rules can cause the router to become overloaded, usually causing operators to ill advisedly remove rules. If an organization's routers are on the edge, minimizing this objective function could be important:

$$\max_{sn1, sn2 \in Subnet} \left| \bigcup_{i \in Interface} rules(sn1, sn2, i) \right|$$

There are many other objective functions that can be defined. This list is just meant to be illustrative.

## 4 Implementation Experience

To test the usefulness of this approach we wanted to apply the model to a realistic test case. In this test case we defined a configuration consisting of 26 services in 17 deployable units, with 8 different service interfaces, deployed on 160 servers in 8 different server classes running on 5 subnets. The availability of the servers varied from 3 nines of availability (i.e. 99.9%) to four nines. The service availability requirements of the top-level services varied from three to four nines. The derived service availabilities for the deeper services went up to five nines and these deeper services were constrained as needing a security distance from the client of at least 3. We set the objective function to maximize the minimum level of over capacity from among the 17 deployable units.

The services were designed to model a modern multi-tier web based system consisting of client accessible static content and reverse web proxy services fronting for an inner tier of application services providing the application UI control and page generation. The UI services were then built on a tier of business logic services. Unlike the other service layers, the business logic services are

available both from the proxy layer, the UI layer and itself. The business logic services in turn build on a set of file and database services, which are in turn built on a set of virtual disk services implemented by a storage area network.

Considerable tuning in the search procedure was needed to order the configurations tested so that the progress towards a solution progressed at a reasonable rate. At this point, OPL is able to find acceptable solutions after running for several minutes on a single 1.5 GHz processor. Finding optimal solutions for non-trivial objective functions is more elusive as the entire solution space often has to be searched, taking over 10 hours for the sample problem. For many uses this performance is adequate, for example, in configuring an enterprise data center for a new application or an application service provider for a new customer. For other uses, such as online reconfiguration after a device failure or configuring a dynamic grid computer, this performance is not adequate.

Note that a numerical instability in the availability computations currently limits the number of servers per server class to 21. This result indicates that this approach to solving configuration problems is promising; though much more work remains to be done to show that it is practical and efficacious.

## 5 Related Work

A modelling based approach to quality of service prediction is standard fare in queuing theory, but the focus is generally on the much more difficult measure of response time, a measure we leave out of our analysis because of its complexity. However the typical server graph used in queuing theory carries over to the dependency graph used here.

Other attempts have been made to model quality of service properties of distributed systems, most recently in the context of a service grid [5], but many fewer properties are being optimized for. Other current work on service grids is focused on mechanism of configuration rather than the optimization of configurations [6].

The most closely related work to this has been done in the area of provisioning of storage in a storage network. Data storage and services are closely related, and in fact one can think of data access as a special case of service provisioning, where it happens that the services allow for data access. Work done in this area includes innovative work done at HP [6–9] in configuring storage systems. The authors have made their own forays into storage management in [4, 10].

Other related work lies in network provisioning, where resources needed to provide the required quality of service are reserved in advance. In this work the model is more based on dynamic load rather than the static load model used in this work. Examples include [11, 12].

A subset of the service provisioning problem being considered here was addressed using constraint satisfaction in [13], but the problem was simple enough that the big guns of constraint satisfaction was not necessary for the solution.

The SmartFrog [8] system from HP provides tools for describing and deploying configurations.

## 6 Conclusion and Future Plans

The approach of producing an abstract model of a complex system and reasoning about that abstract model is an oldie but a goodie. In this paper we have applied this technique to the problem of configuring a service oriented architecture. We have shown how to compute properties of the resulting network and to use those properties to drive the automatic optimization of that network to meet a set of requirements defined over those properties.

A necessary future step for this research is to experiment with configuring real systems to verify that the promised gains are actually achievable. This can also be used to determine if there are constraints missing in our model that allow the production of flawed configurations.

Another area for extension is the development of new types of security measures. Our security distance metric can be refined by allowing each routing rule to have a separate breakage cost, instead of the unit cost used here. The attacker would search for the lowest cost path to the inner systems. In addition the rules can be arranged in a partial order to represent which rules are implicitly broken when another rule is hacked. This can be used to model the fact that once a successful attack on a system is found, the same attack can be used against similar systems with no additional cost.

In this paper we define availability as a service having enough available resources to perform its function. This definition does not mean that the service has those resources for a long enough contiguous interval of time to actually *perform* its function. For example, a diabolical highly available server with a very short MTBF but an incredibly small MTTR, may provide high availability using our definition, but unacceptable performance in real situations. We would like to define service availability as the probability that a given request is successfully processed however, this doesn't easily match up with the definition of availability for a server, which is necessarily time based.

## Acknowledgements

The authors would like to thank the anonymous referees for their helpful comments on this paper. The authors would also like to acknowledge the support provided by the Boole Institute for Research in Informatics (BCRI), for the work included in this paper.

## References

1. Graham, S.L., Kessler, P.B., McKusick, M.K.: gprof: A call graph execution profiler. In Thomas, W., ed.: Proceedings of the SIGPLAN '82 Symposium on Compiler Construction. Volume 17(6) of SIGPLAN Notices., Boston, MA, USA, ACM Press (1982) 120–126
2. Herrold, R.: Rpm package manager (2002) `http://www.rpm.org`.

3. Elizabeth D. Zwicky, Simon Cooper, D.B.C.: A Handbook of Process Algebra. 2 edn. O'Reilly (2000)
4. Aziz, B., Foley, S.N., Herbert, J., Swart, G.: Configuring storage area networks for mandatory security. In Farkas, C., Samarati, P., eds.: Proceedings of the 18th IFIP Annual Conference on Data and Applications Security, Sitges, Catalonia, Spain, Kluwer (2004) 357–370
5. Al-Ali, R., Hafid, A., Rana, O., Walker, D.: An approach for qos adaptation in service-oriented grids. Concurrency Computation: Practice and Experience **16** (2004)
6. Alvarez, G.A., Borowsky, E., Go, S., Romer, T.H., Becker-Szendy, R., Golding, R.A., Merchant, A., Spasojevic, M., Veitch, A.C., Wilkes, J.: Minerva: an automated resource provisioning tool for large-scale storage systems. ACM Transactions on Computer-Systems **19** (2001)
7. Anderson, E., Hobbs, M., Keeton, K., Spence, S., Uysal, M., Veitch, A.C.: Hippodrome: Running circles around storage administration. In Long, D.D.E., ed.: Proceedings of the FAST'02 Conference on File and Storage Technologies, Monterey, California, USA, USENIX (2002) 175–188
8. Goldsack, P., Guijarro, J., Lain, A., Mecheneau, G., Murray, P., Toft, P.: Smartfrog: Configuration and automatic ignition of distributed applications. In: Proceedings of the HP OpenView University Association 10th Workshop, University of Geneva, Switzerland (2003) `http://www.smartfrog.org/papers/SmartFrog_Overview_HPOVA03.May.pdf`.
9. Ward, J., O'Sullivan, M., Shahoumian, T., Wilkes, J.: Appia: automatic storage area network design. In Long, D.D.E., ed.: Proceedings of the FAST'02 Conference on File and Storage Technologies, Monterey, California, USA, USENIX (2002) 203–217
10. Swart, G.: Storage management by constraint satisfaction. In: Proceedings of the Workshop on Immediate Applications of Constraint Programming, Kinsale, Cork, Ireland (2003)
11. Balter, R., Bellissard, L., Boyer, F., Rivelli, M., Vion-Dury, J.: Architecting and configuring distributed applications with olan. In: Proceedings of the 1998 IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing. Volume 1518 of Lecture Notes in Computer Science., The Lake district, UK, Springer Verlag (1998) 241–256
12. Chen, S., Nahrstedt, K.: An overview of quality-of-service routing for the next generation high-speed networks: Problems and solutions. IEEE Network Magazine **12** (1998) 64–79
13. Martn-Díaz, O., Cortés, A.R., Durán, A., Benavides, D., Toro, M.: Automating the procurement of web services. In Orlowska, M.E., Weerawarana, S., Papazoglou, M.P., Yang, J., eds.: Proceedings of the 1st International Conference on Service-Oriented Computing. Volume 2910 of Lecture Notes in Computer Science., Trento, Italy, Springer Verlag (2003) 91–103