Automatic Configuration of Services for Security, Bandwidth, Throughput, and Availability

Garret Swart*
University College Cork
Cork, Ireland
g.swart@cs.ucc.ie

Benjamin Aziz
University College Cork
Cork, Ireland
b.aziz@cs.ucc.ie

Simon Foley
University College Cork
Cork, Ireland
s.foley@cs.ucc.ie

John Herbert
University College Cork
Cork, Ireland
j.herbert@cs.ucc.ie

ABSTRACT

The process of efficiently deploying a complex system of services on a complex network of servers is tedious and error prone, with many properties to check and many possibilities to examine. Automated tools are needed to turn this into a humanly tractable problem. We present a precise model of a service-oriented computing system that allows many important configuration properties to be defined and optimized for, including throughput, network bandwidth, security and availability. We transform this model into a system of constraints that can then be solved using mathematical and constraint programming yielding an optimal system configuration that meets all the stated requirements. We have implemented this in OPL and have used it to generate optimal configurations for realistic systems with tens of services running on hundreds of servers communicating on multiple subnets.

Categories and Subject Descriptors

C.2.0 [Computer Communication Networks]: General – Security and protection. C.2.3: Network Operations – Network management, C.4.0: [Performance of Systems] Modeling techniques, Performance attributes K.6.4: [Management of Computing and Information Systems] System Management – Quality assurance

General Terms

Management, Performance, Security.

Keywords

Automated Server Provisioning, Service-oriented Computation, Automated Network Management, Autonomous Computing, Quality of Service (QoS)

1. INTRODUCTION

The process of deploying complex systems of services on a complex network of servers is fraught with peril for the systems administrator. No one wants to be on the front page of the New York Times when a multi-million dollar system becomes inaccessible due to an unforeseen problem with an obscure configuration setting. No one wants to be asked by the boss why the network has to be upgraded when a simple reconfiguration might be adequate. No one wants to spend days looking at ways of making do with the current hardware because the budget is used up but new applications are coming online. Good systems configuration needs precise checking and exhaustive optimization, not just rules of thumb and heartfelt prayer.

We suggest solving this problem by the use of mathematical modeling. Methods rooted in mathematics can result in configurations that are provably optimal and correct. Methods rooted in mathematics can be amenable to machine reasoning rather than the more onerous and error prone human variety. Using formal techniques we can translate engineering intuition into mathematical constraints.

In this paper we define a precise model of a service-oriented computing system. We argue that this model is close enough to reality to be interesting. We then define various properties of the model that correspond to important properties in real systems. The properties that we define and optimize for are network security, server throughput, service availability and network bandwidth. All of these terms have been bandied about enough that they need careful definitions, and we define them using our mathematical model.

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 containing 26 services and 240 servers on 5 subnets.

Novel aspects of this work include:

– The application of configuration modeling and optimization to general service oriented computation. The service model that we introduce and the complex relations that services can have with each other allows us to model service-oriented systems "from the phosphor to the oxide."

– The simultaneous modeling of many configuration properties so that the values of these properties can be played off against each other, and a framework that allows even more properties to be defined and modeled.

*This author's current address is: IBM Almaden Research Center; 650 Harry Road; San Jose, California 95120 gswart@almaden.ibm.org.

– 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.

Modeling and optimization, while at the core of automated configuration, is not all that is needed. Other aspects of this problem include:

– Model reverse engineering: Generating a model from a real system is a huge task. System administrators need tools to facilitate generation of system requirements from a real system that is known to meet its requirements. The resulting model will allow the search for cheaper configurations meeting the same requirements or new configurations meeting adjusted requirement, E.g. Build me a system that behaves just like my old system but that handles twice the load at half the cost.

– Requirements understanding: A complex system may have thousands of requirements. How can we ensure that all the important requirements for the correct working of a system have been captured? Missing a requirement may produce a system that, while it meets all of the stated requirements, does not function. The implications of security requirements are notoriously difficult to understand but we want to ensure that whatever language is used for these requirements supports the WISIWIM requirement: "What I Said Is What I Meant."

– Configuration deployment: Once a new configuration has been determined, how can we reliably implement the change from the current configuration to the new configuration? This may involve rewiring network connections, reconfiguring routers, redeploying services on different servers. Each task should be automated if possible, and in any case checked for correct completion.

– Model refinement: Any abstract model is just that, a model. It is meant to mirror reality and any place where the model does not mirror reality should be flagged and fixed. For example, the model may predict that adding a processor in a particular role will improve performance by 20%. If deploying the change unexpectedly results in a performance improvement of 10% or 50%, we want to adjust our model so that properties of future configurations will be more accurately predicted. When many changes are made simultaneously figuring out where the model should be changed can be difficult.

– Adaptability: A static service configuration is unlikely to stay unchanged for long. New services are being introduced and the properties of existing services change based on market success, developing usage patterns, and service implementation changes. We want to find an optimal sequence of configurations from the current optimum, based on one set of assumptions and requirements, to a new optimum based on a new set of assumptions and requirements.

In this paper we focus on the necessary first step: the modeling and searching for static configurations; other aspects are left to future work.

Service configuration modeling is important to designers of services because it can impact the way they think of their services and the information they need to specify about them. It is important to vendors of service software because it allows complex networks of service software to be installed by the users themselves, rather than by teams of warring vendor consultants. It is important to service-oriented middleware providers, as they will need to provide the tools that provide the facilities listed above. Finally, it is important to those who deploy services, as they will be able to save money and avoid worry as they deploy their service networks.

2. MODELING A SERVICE-ORIENTED SYSTEM

The art of modeling lies in figuring out what to put in and what to leave out. Putting too much in the model results in a model that can be intractable for both humans and machines, while leaving too much out of the model makes important questions impossible to state. The goal of this paper is to leave enough in the model to be able to do realistic network capacity planning, server capacity planning, network security planning and service level availability planning. Anything that is not required to meet those requirements, we left out. 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 will produce. 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 Fig. 1.

Service. The fundamental system component in a service-oriented architecture is, of course, the service. In this context 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.

Implicitly associated with a service is that service's implementation. This implementation, while invisible to the user of the service will determine certain properties of the service, e.g. the load caused by performing an operation or the set of services called by this service.

We denote the set of all services being modeled as a set named *Services*.

To specify the load generated by an invocation of a service, each service must be provided with a per invocation load amount that must be provided by the servers assigned to run this service. This is the expected amount of load caused by handling a single call on the service. The units of this cost might be Java Virtual Machine instructions or any other reasonable unit of execution cost that adequately represents the utilization of resources on a machine running the service. As processor designers, compiler writers, or even marketing directors will tell you, there is no one number that can represent a server's capacity or an application's load, but for this purpose we'll assume that we have one that provides an adequate estimate.

Formally we define a function *loadU* that defines the expected load units caused by a single invocation of the service.

$$loadU : Service \rightarrow \Re^{+}$$

Service Interface. Each service implements a certain protocol or language to facilitate communication with it and other similar services. 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. Note that each service implements only a single interface; we will see below that the concept of a deployable unit, which represents an aggregation of services, provides for the functionality of allowing multiple interfaces to be implemented by a single service.

$$implements : Service \rightarrow ServiceInterface \; .$$

We do not consider the process of how service interfaces may be mapped to services using other services like LDAP or UDDI or how interfaces might be discovered using an ontology.

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 behavior of a particular installation.

For example, Travelocity's car rental service may invoke the Hertz and Avis services an average of 1.4 times per invocation, while LastMinute's car rental service may invoke the Hertz service 1.7 times and the Avis service 2.2 times per call.

As in this example, 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 $dependency1$.

$$dependency1 : Service \times Service \rightarrow \Re^{+}$$

For example, $dependency1$(Travelocity, Avis) = 2.2 as each call to Travelocity results in the Travelocity service calling the Avis service an expected 2.2 times.

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 $dependency1$ matrix. We do this to allow for a concise definition of the service dependency information and because collecting multi-level information of this sort in a distributed system is quite difficult. However if the complete graph has been measured, it should certainly be used in preference to the estimate.

$$dependency : Service \times Service \rightarrow \Re^{+}$$

This is the same approach used in gprof in estimating call graph values [10].

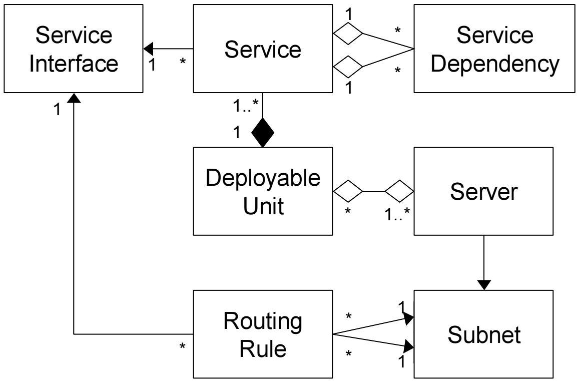

Fig 1: A UML Diagram of the Service Oriented Model

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 behavior 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*.

$$client \in Service$$

Harking back to our car rental example, we can define the client service as making 400 calls per minute on the Travelocity service, 300 calls per minute on the LastMinute service, 100 calls per minute on Hertz and no calls on Avis. Assuming that the Hertz and Avis services make no service calls themselves, then taking the five services in the order: *client*, Travelocity, LastMinute, Hertz and Avis, we have a first level dependency matrix given as:

$$\begin{pmatrix} 1 & 400 & 300 & 100 & 0 \\ 0 & 1 & 0 & 1.4 & 1.4 \\ 0 & 0 & 1 & 1.7 & 2.2 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}.$$

The ones on the diagonal can be thought of as meaning that a call on a service results in one call on itself. We then take the transitive closure of this matrix to estimate the full dependency matrix. The top row of this dependency matrix gives us the frequency count of invocations of the corresponding service during the one-minute client interval:

$$\begin{pmatrix} 1 & 400 & 300 & 1170 & 1220 \end{pmatrix}.$$

These invocation counts can be combined with the preceding service load units to define the total execution load on the system. For example, if $loadU$(Travelocity) = 1000, $loadU$(LastMinute) = 1200, $loadU$(Hertz) = 6000, and $loadU$(Avis) = 7000, then the execution resource needed to meet the throughput requirements on the services are 400,000, 360,000, 7,020,000, and 8,540,000 units per minute respectively, and the total execution rate in the system is 16,320,000 units per minute.

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.

$$availableToClient : Service \to [0,1]$$

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 is 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 [11] 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.

$$deploys : Service \to Deployable\ .$$

The composition of a deployable unit out of a set of services, modeled by this function, could have been generalized to a many to many relationship between services and deployable units, allowing a single service to be implemented by many different deployable units. In practice, practical engineering considerations cause administrators to avoid running more than one implementation of a single service, as this raises the probability of a failure caused by unexpected interactions between different deployable units running different implementations but implementing the same service. However, a service interface is typically implemented by many different services and thus potentially many deployable units.

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:

$$serverAvailability : Server \to [0,1]\ .$$

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*. For example, a large multiprocessor server may be able to perform 50,000 load units per minute while a small server may only be able to support 1000 units per minute. We specify the execution rate of a server with the function *powerU*:

$$powerU : Server \to \Re^{+}$$

As stated earlier, a single power rating per server is a simplification. A more careful model may allow for service specific load ratings.

Resources on servers are assigned by the configuration system to deployable units. A 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:

$$allocU : Deployable \times Server \to \Re^{+}\ .$$

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:

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

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. We assume that each server is assigned to exactly one subnet. The assignment of servers to subnets will typically be an input to the configuration optimization process, though in other circumstances the assignment and creation of subnets might be an output of the process. We also distinguish the subnet from which client communications originate. This will typically represent the public Internet.

Formally, a subnet is just a set of items, *Subnet*, and a function subnet that assigns servers to subnets.

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

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.

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

3. 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. One interesting property that we do not define is latency. Reasoning about latency is a very tricky issue and we do not attempt to deal with it here.

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.

$$\begin{aligned} & availableToClient(s1) \\ & \quad \leq serviceAvailability(s1) \\ & \quad\quad \leq \prod_{\substack{\forall s2 \in Service \\ dependency1(s1,s2)>0}} serviceAvailability(s2) \end{aligned}$$

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. Disconnecting availability and throughput admits of an implementation of a highly available system that while it may be nominally available, failures may have degraded its throughput so much that it may be considered dead by its users. One could complicate matters by defining multiple levels of availability, e.g. there is 99% probability that full throughput is available, but 99.99% probability that 50% throughput is available. But we do not do so in this paper.

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:

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

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 $S \subseteq Server$ 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:

$$\begin{aligned} setProbability(S) = & \prod_{\forall serv \in S} serverAvailability(serv) \\ & \times \prod_{\forall serv \in Server - S} (1 - serverAvailablility(serv)) \end{aligned}$$

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:

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

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:

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

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

$$\begin{aligned} &\forall d \in Deployable, \quad deployAvailable(d) \\ &\quad \geq \max_{\forall s: d = deploy(s)} serviceAvailability(s), \end{aligned}$$

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 and 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 Fig. 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. Note the NOC subnet, used for monitoring and administration, has connectivity to all the other subnets and if misconfigured can provide a shortcut access path into the deepest levels of the system, e.g. if servers in the NOC can be accessed from the Internet, sometimes allowed so that administrators can work from home.

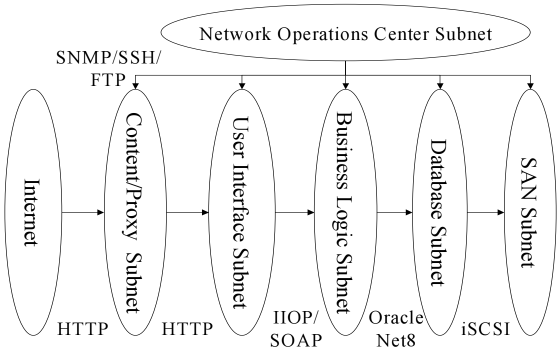

Fig. 2: Typical Subnet Structure

One way to increase the security level of a service to infinity is to make it inaccessible to clients by only running the service on servers that are on subnets with no direct or indirect path from the client. This makes these services secure, but unfortunately, also unavailable to the rest of the public.

When configuring routing rules it is important to allow communication between subnets where it is needed, e.g. services running on those subnets need 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 [16].

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

$$\begin{aligned} &\forall s1, s2 \in Service : dependency1(s1,s2) > 0 \Rightarrow \\ &\forall serv1, serv2 \in Server : \\ &\begin{pmatrix} allocU(deploys(s1), serv1) > 0 \\ \wedge\, allocU(deploys(s2), serv2) > 0 \end{pmatrix} \\ &\Rightarrow interface(s2) \in rules(subnet(serv1), subnet(serv2)) \end{aligned}$$

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 *securityDistance*(*s*1*, s*2), 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.

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

Then we can define the security distance with the following recurrence:

$$\begin{aligned} &securityDistance(s1,s2) = \\ &\quad \begin{cases} 0, \;\; \text{if } connected(s1,s2) \\ 1, \;\; \text{if } dependency(s1,s2) > 0 \wedge \neg connected(s1,s2) \\ \min_{\forall s3 \in Service} \{ securityDistance(s1,s3) \\ \qquad\quad + securityDistance(s3,s2) \}, \quad \text{otherwise} \end{cases} \end{aligned}$$

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 by this paper's authors [5], 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 was not used in the OPL implementation described in section 5, but was used in a separate OPL model described in [5].

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*(*sn*1, *sn*2, *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 $s2$.

4. 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. The cost of a system is generally a linear formula involving the number and cost of each server and perhaps the number of subnets. Meeting requirements with the least cost is a common objective in optimization. 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:

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

– 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:

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

– 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.

5. IMPLEMENTATION EXPERIENCE

The formulas given here can be entered nearly unchanged to build an OPL, Optimization Programming Language, application. In this section we describe a few changes that were made to facilitate expressing these formulas in OPL and how we reduced the search space to get faster results. We also describe a realistic example that we used to test the feasibility of the approach.

5.1 OPL Model Changes

A first implementation of the OPL model was made that used the functions as defined here directly. Unfortunately this model did not scale to larger configurations. The primary problems were the large solution space for the *allocU* function, the non-linearity in the probability function *setProbability* and the difficulty in entering the large amount of data for hundreds of nearly identical servers.

To speed up the search and ease the data entry we made two simplifications to the model given above:

– We assume that the allocation of load units to a deployable unit is identical on all of the servers it is assigned to. This reduces the search space considerably without removing too many interesting solutions. This is an acceptable simplification as a big disparity between allocations for the same deployable causes difficulties in reaching availability goals as the failure of the server with the largest allocation for a given deployable causes an undue loss for that deployable, making achieving availability much more difficult.

– We group the servers into server classes of identical servers; all the servers in a class have the same availability, power units and are on the same subnet. Each server class might correspond to a rack of servers in a data center.

– We assign deployable units only to servers in the same server class. This simplifies the computation of the *runsIn* function above, makes the security limits easier to reach, and reduces the solution search space considerably, again without removing many important configuration possibilities. Such a server assignment is typical of service assignments inside a cluster in a single data center. Services that are replicated across a WAN and logically run on different subnets must be modeled as separate services implementing the same service interface. In most cases this is appropriate as, given WAN communication costs in general and the costs of single copy serializablility in specific, the services are not really identical and are assigned to data centers manually.

By making these changes we can greatly simplify the computation of the products in the *setProbability* function by making use of the binomial theorem. We can do this because each *serverAvailability* probability used for a deployable is identical, so instead of summing over the power set of servers available, we sum over the number of servers available. Similarly the function *allocSU* is simplified because the *allocU* function values are either identical or zero for each server being used by a deployable unit.

For a particular deployable unit *d* that is assigned to *deployU*(*d*) units on *dcnt*(*d*) servers all in a server class with availability *avail*, the probability that the system is available with enough resources is:

$$\sum_{\left\lceil \frac{reqLoadU(d)}{deployU(d)} \right\rceil \le w \le dcnt(d)} \binom{dcnt(d)}{w} avail^{w} (1-avail)^{dcnt(d)-w} .$$

5.2 OPL Implementation

An interesting feature of the OPL programming model is the six different sub-languages it supports, five of which are used in this application:

– A data declaration language that is used to define the form of the input data, the intermediate data, and the search space of the variable data.

– A sequential initialization language that is used to compute the values for the intermediate data based on the input data. Intermediate data so computed is considered to be ‘ground’ and can be used in more contexts than the variable data. In our application this is used to compute *dependency* from *dependency1* and to compute *serviceAvailability* based on *dependency1* and *availableToClient*. The former is straight forward, but the latter involves spreading probabilities.

The approach taken is to consider each target service in an order compatible with the *dependency*1 relation and assign that service a computed availability. We do this by looking at all the up-level services that the target service is used by and taking the maximum availability requirement implied by each such up-level service. If the up-level service has an availability requirement either specified or computed, we then look at all of the subsidiary services used by the up-level service. Some of those subsidiary services may precede the target service in the ordering and thus may already have a computed availability, these availabilities are divided out of the target availability. The remaining probability is spread evenly among those without specified availability by taking the appropriate root of the availability.

– A data instantiation language that is used to provide the input data and thus define a particular instance of the model to be solved. Data items instantiated in this language are stored in a separate file from the other items, facilitating using the same model for many different similar problems.

– A first order logic based constraint language used to specify the constraints relating the input data, the intermediate data and the output data.

– A backtracking based search control language that is used to control the search through the output space. This language is the most problematic as this has to be modified when a new optimization criterion is chosen. A subtle change in strategy can make the difference between search that won’t end for many lifetimes and a result that is generated in minutes.

The commented OPL application is available on the authors’ web site [13].

5.3 Realistic Test Case

To test the usefulness of this approach we wanted to apply the model to a realistic test case. The problem is NP-hard, so one can certainly find problem instances which cannot be solved in a reasonable amount of time. However we wanted to pick a test case that might come up in practice and see how this approach worked on this example.

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

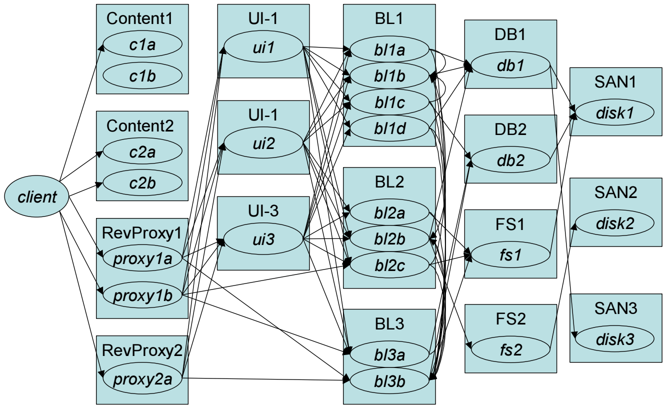

Fig. 3: Test Case Services, Deployables and Dependencies

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 and from the UI layer and they have a rich degree of interconnection that is difficult to see in the diagram. The services have been factored to remove any cycles from the dependency graph. 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. The services, their grouping into deployable units and the dependencies are illustrated in Fig. 3.

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 nontrivial 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.

6. RELATED WORK

A modeling 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 [1], 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.[2]

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 [2][3][9][15] in configuring storage systems. The authors have made their own forays into storage management in [5][12].

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 [6][7].

A subset of the service provisioning problem being considered here was addressed using constraint satisfaction in [8], but the problem was simple enough that the big guns of constraint satisfaction was not necessary for the solution.

The SmartFrog [9] system from HP provides tools for describing and deploying configurations.

7. 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 services on a network of servers. 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.

There have been advances in constraint systems and automated reasoning in recent years and using these techniques to design and maintain complex computing systems is an opportunity ready to be grasped.

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. Here we explore a simple security distance metric, but many other types of threat can be defined, e.g. the information risk measure used in [5]. 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. We are looking into this issue.

Incremental configuration, finding the minimal reconfiguration of an existing system to adapt to a new set of requirements or a new environmental condition, is also a focus for future work.

8. ACKNOWLEDGEMENTS

Thanks to Aad van Moorsel and the anonymous referees for their insightful comments.

9. REFERENCES

[1] Rashid Al-Ali, Abdelhakim Hafid, Omer Rana1 and David Walker, An Approach for QoS Adaptation in Service-Oriented Grids, Concurrency Computation: Practice and Experience, 2004, 16(5)

[2] G.A. Alvarez, E. Borowsky, S. Go, T. H. Romer, R. Becker-Szendy, R. Golding, A. Merchant, M. Spasojevic, AVeitch, and J. Wilkes. Minerva: an automated resource provisioning tool for large-scale storage systems. ACM Transactions on Computer Systems, November 2001.

[3] E. Anderson, M. Hobbs, K. Keeton, S. Spence, M. Uysal, and A. Veitch. "Hippodrome: Running Circles around Storage Administration." In Proceedings of the Conference on File and Storage Technologies, January 2002.

[4] Argonne National Laboratory. The globus project. See Web Site at: http://www.globus.org/

[5] B. Aziz, S. Foley, J. Herbert, and G. Swart. "Configuring Storage Area Networks for Mandatory Security." In Proceedings of the IFIP WG 11.3 Working Conference on Data and Application Security, July 25-28, 2004.

[6] R. Balter, L. Bellissard, F. Boyer, M. Riveill and J.Y. Vion-Dury, "Architecting and Configuring Distributed Applications with Olan", In Proc. IFIP Int. Conf. on Distributed Systems Platforms and Open Distributed Processing (Middleware'98), The Lake District, 1518 September 1998.

[7] Shigang Chen and Klara Nahrstedt. An Overview of Quality-of-Service Routing for the Next Generation High-Speed Networks: Problems and Solutions, IEEE Network, Special Issue on Transmission and Distribution of Digital Video, Nov./Dec. 1998.

[8] O. Martín-Díaz, A. Ruiz-Cortés, A. Durán, D. Benavides and M. Toro, "Automating the Procurement of Web Services" In Proceeding ICSOC 2003. LNCS. Springer Verlag, 2003.

[9] Patrick Goldsack, Julio Guijarro, Antonio Lain, Guillaume Mecheneau, Paul Murray, Peter Toft, "SmartFrog: Configuration and Automatic Ignition of Distributed Applications," 2003 HP Openview University Association conference, See http://www.hpl.hp.com/research/ smartfrog/papers.

[10] S.L.Graham, P. B. Kessler, M. K. McKusick, "gprof: A Call Graph Execution Profiler" In Proceedings of the SIGPLAN '82 Symposium on Compiler Construction; SIGPLAN Notices; Vol. 17, No. 6, pp. 120-126, June 1982.

[11] RPM Package Manager. http://www.rpm.org.

[12] G. Swart, "Storage Management by Constraint Satisfaction," In Proceedings of the Workshop on the Immediate Applications of Constraint Programming held as part of the Ninth International Conference on Principles and Practice of Constraint Programming (CP'03), 29 September 2003.

[13] G. Swart, "Backup material for Automatic Configuration of Services." http://www.cs.ucc.ie/~gs1/ServiceConfig.

[14] Pascal Van Henteryck. OPL; Optimization Programming Language. The MIT Press, 1999.

[15] Julie Ward, Michael O'Sullivan, Troy Shahoumian and John Wilkes, "Appia: automatic storage area network design." In Proceedings of the Conference on File and Storage Technologies, January 2002.

[16] Elizabeth D. Zwicky, Simon Cooper, D. Brent Chapman *Building Internet Firewalls*, O'Reilly & Associates; 2nd edition (January 15, 2000)