Optimization of E-Service Solutions with the Systems of Servers Library

Vadim Kotov, Holger Trinks
Computer Systems and Technology Laboratory
HP Laboratories Palo Alto
HPL-2000-54
April, 2000

E-mail: kotov@hpl.hp.com
htrinks@hpl.hp.com

e-service systems, system integration, system modeling, system analysis and optimization, Communicating Structures Library, services, servers, service partitioning, load balancing, genetic algorithms

The report describes a modeling library for analysis and optimization of distributed service systems: enterprise computing infrastructures, E-commerce and E-service systems. The library uses a small number of basic notions, such as *service, server, cluster* (of servers), *clients* and *message*, which allow us to describe, analyze and optimize various system configurations and deployment of services among servers. The service requests and responses are modeled by messages traveling in the system and using the system common resources, such as network bandwidth and cache space. Clients send their requests for services to servers, which either return responses or may issue secondary requests for other services. The messages can form complex activities such as *transactions* (*sessions*). To find optimal deployments of services among servers, different partition policies can be used including partitions based on genetic algorithms.

The *Systems of Servers* is built on top of the Communicating *Structures Library* (CSL), a basic library for modeling large-scale distributed systems [1,2]. In its turn, it is a base for more specialized libraries that target specific classes of service systems. All these libraries are part of the *System Factory* [4], a modeling environment for system integration and customization.

1 INTRODUCTION

Distributed services and servers form the backbone of modern computing environments based on inter- and intranets: enterprise-computing infrastructures, WWW, E-commerce, E-service systems, etc.

Though the diversity of service types and patterns of interaction between clients and servers is quite large, it is possible to extract a relatively small number of basic concepts, objects, and "communication templates" that allow us to adequately model the typical situations and problems emerging in distributed server environments, to analyze these problems, and then to solve them.

The term E-services is currently used in commercial context mostly to distinguish internet-based, end-to-end, customer-facing business services from traditional bricks-and-mortar operations. In this report, we treat an E-service, or just a *service*, as a more general and more technical notion, namely as an activity in a system that

- is provided by a server,
- is requested by clients or by servers that need another service to satisfy the original service request (secondary service),
- consumes some system resources both in servers and in the system infrastructure,
- competes for these resources against other services.

To request a service, a client issues *messages* that travel in the system and use common system resources, such as network bandwidth and cache space. A server may either return a *response* or issue secondary requests for other services that are needed to fulfill the primary service. Messages may form more complex activities called *transactions* or *sessions.*

One server may host many services. Servers may be grouped into clusters of servers. Services also may be grouped forming a hierarchy of services and subservices. One service may be partitioned and replicated among many servers and among many clusters of servers.

A service system works successfully if it is able to deliver services in due time, requires minimal resources, is highly available, secure, and satisfies other quality of service requirements. Competition for resources and traffic bottlenecks may cause significant system performance degradation. This requires special attention to the service deployment and to the message traffic management.

With increasing demand on services and their assortment, the variety of the service system architectures and their complexity is growing rapidly. The difficult problem, which service system designers face, is the selection of good solutions in this large solution space. The modeling library *Systems of Servers* described in this report is intended to help them to do that. It defines a generic set of relatively simple high-level abstractions. They can be used to conveniently describe various service system architectures, different service patterns and to introduce metrics for quality of services and arrange quantitative search for optimal solutions. The Systems of Servers library is built on top of the *Communicating Structures Library* (CSL) developed for modeling and design of large-scale, massively distributed systems [1]. In its turn, Systems of Servers is a base library for more specialized libraries that target specific classes of E-service systems.

The report starts with two examples of E-service systems (Section 2). The first example is a multi-tiered electronic delivery system. The second example is an E-service system based on the E-speak paradigm. The main principles and objects of the Communicating Structures Library are described in Section 3; basic objects of Systems of Servers are introduced in Section 4. To find optimal deployments of services among servers, different partition policies can be used. Some of them are described in Section 5. Load balancing and caching are briefly addressed in Sections 6 and 7.

2 EXAMPLES OF E-SERVICE SYSTEMS

2.1 Multi-Tier Electronic Delivery Systems

The use of electronic delivery systems (EDS) has grown tremendously since e-mail and the World Wide Web have emerged as common and convenient infrastructures for communication of customers with businesses. These developments are transforming the market for electronic delivery. A typical EDS consists of up to 4 tiers (Figure 1):

- Tier 1: *Thin clients.* They are external Internet, ATM, and phone/fax clients accessing the system for services via transactions.
- Tier 2: *Front-end (Web) services and servers*. This tier provides a secure foundation for the applications that Internet-enable an enterprise. It connects the outside clients to the inside servers via an internal secure boundary.
- Tier 3**:** *Application services and servers*. This tier provides self-service solutions deployed at several application and DB servers. Transactions may be packaged into modules that access the services.
- Tier 4: *Back-end services and servers.*

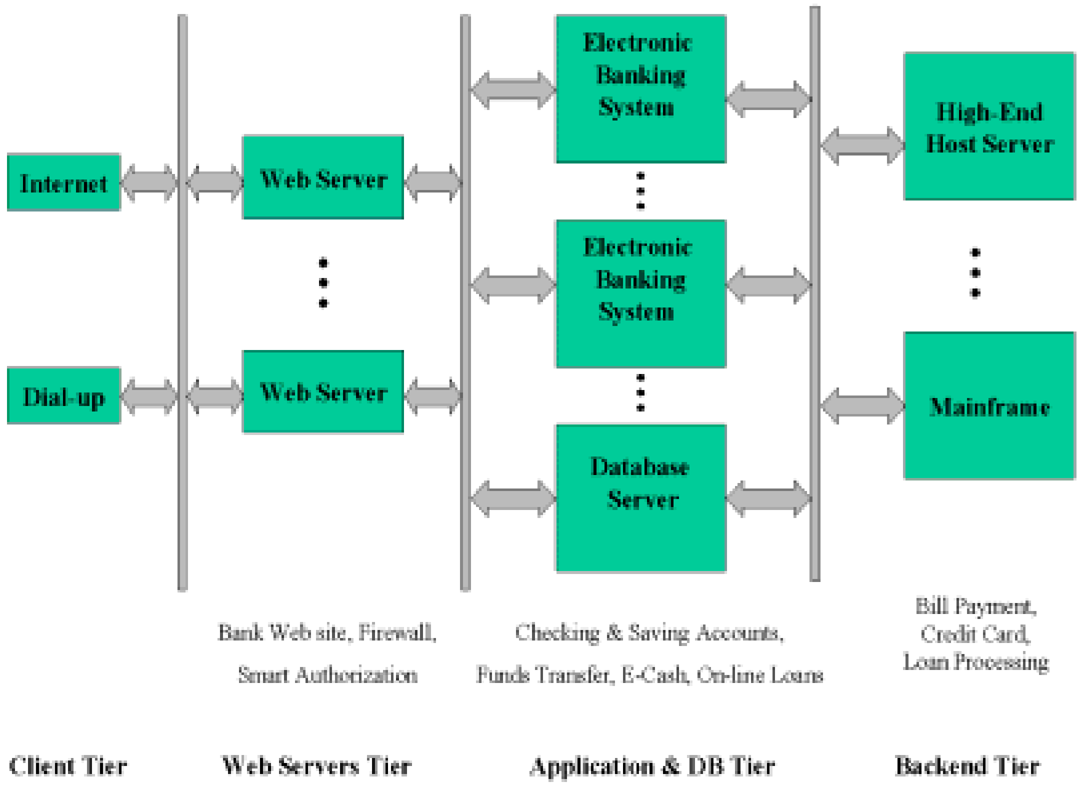

Internet
Dial-up
Web Server
Web Server
Electronic Banking System
Electronic Banking System
Database Server
High-End Host Server
Mainframe
Bank Web site, Firewall, Smart Authorization
Checking & Saving Accounts, Funds Transfer, E-Cash, On-line Loans
Bill Payment, Credit Card, Loan Processing
Client Tier
Web Servers Tier
Application & DB Tier
Backend Tier

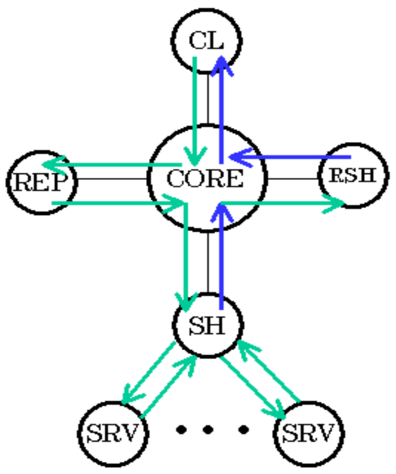
Figure 1

In terms of *Systems of Servers*, this EDS may be presented as a set of services provided by different servers in appropriate tiers that are clusters of servers. Clients from Tier 1 send requests for specific services. These requests are forwarded to Tier 2 where they invoke services of the web servers. These web services issue *secondary requests* to the services provided by the application servers of the Tier 3. An application service may either execute the request and send back a *response* message with the results of the execution or generate a new secondary request for a service at the back-end tier. In the latter case, the last tier generates the final response. It may happen, that some services may generate multiple secondary requests, which, at the end, produce several responses. That means that one initial request may, in general case, create a flow of messages in the system and activate multiple services.

This generic EDS architecture may be modified and customized for specific electronic delivery services in different business segment, such as banking or retail businesses.

2.2 E-speak Infrastructure

E-speak is a "computing utility" computation model, in which services are virtualized. This means that a client may request a service without knowing where it is provided, how actually to access it, or what are implementation details. An E-speak system selects an actual service (an actual server) for the client or proposes a choice of services to select. E-speak is intended to enable ubiquitous services over the network.



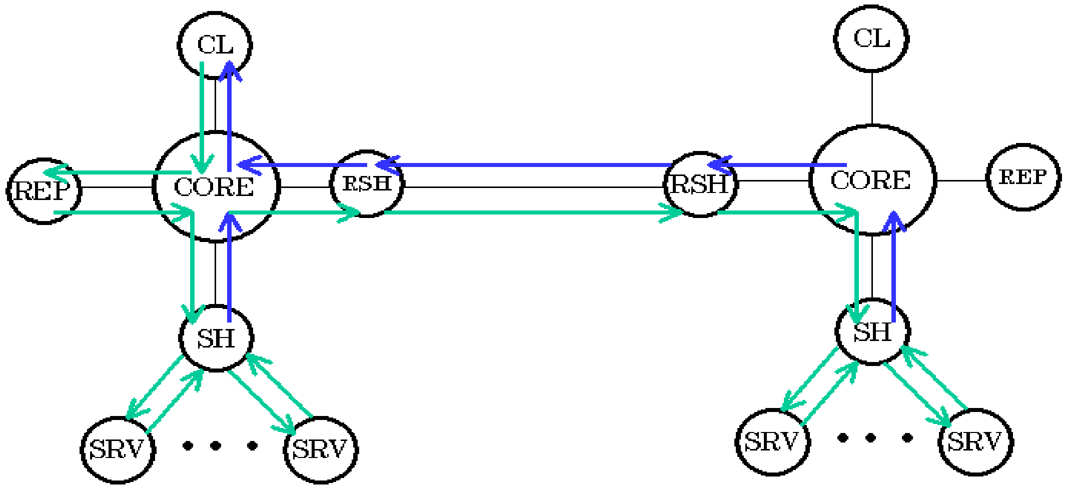
**CL – client, REP – Repository, SH – Service Handler,
RSH – Remote Service Handler, SRV - Server**

Figure 2

Figure 2 illustrates the main components of a typical E-speak system deployment. An E-speak system consists of a set of *Logical Machines*, each including:

- An E-speak *Core* that is the E-speak basic control unit, a “daemon” providing the distributed mediation layer,
- A *Repository*, in which all services managed by the Core are registered,
- A *Service Handler*, that processes requests for local services,
- A *Remote Service Handler* that handles the inter-machine exchange of service requests and responses,
- *Servers* that actually provide requested services.

In order to get a service, a client sends a request message to the local Core. To make a (secondary) request for the service residing at the same logical machine, the Core accesses the Repository to obtain registration information that identifies the Service Handler associated with the requested service. The Core then forwards the requests either to the local or to remote Service Handler. In the first case, the Service Handler transfers requests to Servers that actually provide services. The results of the services are sent back to the Service Handler, the Core and finally to the Client.



CL – client, REP – Repository, SH – Service Handler, RSH – Remote Service Handler, SRV - Server

Figure 3

In the remote case, the local Remote Service Handler passes the request to the peer Remote Service Handler in a remote Logical Machine hosting the requested service. That Remote Service Handler contacts the Core at that remote Logical Machine and then process continues as if this request is local until a response comes back to the Core. Then the Core sends the response to the "local" Remote Service Handler that returns it to the Remote Service Handler and to the Core of the machine at which the request originated. Figure 3 shows the itinerary of messages in the case of remote service access.

In terms of Systems of Servers, the Core, Service Handlers, and Repository all are servers providing services. These "internal" system services are not requested by clients but are present in the system and are invoked to access the "terminal" service requested by the client.

3 COMMUNICATING STRUCTURES LIBRARY (CSL)

The Systems of Systems infrastructure is built in the framework of *Communicating Structures* [1], an abstract system representation that views systems as hierarchical distributed structures composed in a uniform and systematic way. The system components are represented simply as *nodes*. Each node has *memory* that may contain *items*. *Nets* are sets of links that connect the nodes. The items are generated at some nodes and move from node to node along links, with some delay. Nodes may modify items. The item traffic models the data traffic in a system, which is represented as a communicating structure.

Items, nodes, memories, and nets may be elementary or may be aggregate and have some structure. For example, an item may represent a simple data such as a word, a frame, a packet, as well as a complex message with large chunks of data or even a transaction consisting of many messages. The nodes may represent “atomic” units or larger aggregate system components. Nets may represent simple point-to-point links as well as busses, crossbars, interconnects, cascaded switches, LANs, communication lines and WANs.

Communicating Structures are high-level, template-style representations of systems in the same sense as template vectors represent vectors of anything. They do not fix behavioral or implementation semantics of basic objects, attributes, functions and processes, but rather provide default semantics for them. That makes possible to “emulate” different conceptual models, such as simulation models, queuing networks, or Petri nets, in the framework of a common abstraction basis.

Communicating Structure Library (CSL) [2] is an object-oriented implementation of Communicating Structures. The CSL objects may be assigned different attributes (numbers, variables, functions, and processes) that serve to refine these objects, customize them, supply them with external data, change their behavior and navigate among different levels of modeling detail.

4 OBJECTS OF SYSTEMS OF SERVERS

The main objects of *Systems of Servers* are:

- services,
- servers,
- clusters of servers (and clusters of clusters),
- clients (proxies),
- messages,
- transactions (or sessions).

Servers, clusters and proxies are refined CSL nodes; messages and transactions are refined CSL items.

Examples of problems that can be addressed in *Systems of Servers* are:

- comparison of topologies of distributed servers,
- partitioning of services among servers and clusters of servers,
- caching strategies,
- queuing and scheduling policies,
- load balancing,
- admission control.

4.1 Services

A *service* is an abstraction that enables the description of a client/server relationship. Each service is described by its *name*, its set of *subservices* (for an aggregate service) and some other attributes that characterize specific features of the service. Hence, services form a tree structure. An aggregate service[1] may be either *real* or *virtual*. Simple services are all real. Real services are assigned to (simple) nodes called *Servers* (see section 4.4). A server may host several (real) services. The same service may be provided by different servers. Virtual services are not assigned to servers. They are used to group services into classes of services and/or to accumulate statistics for a group of services.

The relation between service and subservice may be treated and used in different ways:

For example, a virtual service *Transportation* could have the (real) subservices *Plane, Train* and *Car.* Another virtual service *Lodging* could have the subservices *Hotel, Motel* and *Campground.* A request for the *Transportation* service means actually requesting some of its subservice (with some probability).

It is often convenient to treat a subservice as a secondary service of its parent service. For example, the subservices *"Rent a Car"* and *"Book a Hotel"* could be subservices of the service "*Vacation Arrangement"*. Such an interpretation of the subservice relation is chosen as the default one in the *Systems of Servers* library.

A *Service Matrix* summarizes the information about services requesting other services. The elements of this matrix show the probability with which a service requests another service. The matrix is used to optimize the service deployment and load balancing.

4.2 Messages

A *Message* is a CSL item that carries requests and responses between clients and servers. Each message has information about a service and a tag that indicates whether it is a

[1] a service that is not a leaf in the tree structure

request or a response. Messages containing primary requests for services are always generated by clients. A response message always returns to the client that generated the primary request. Responses to secondary requests return either directly to the client or via the server that issued the secondary request. Due to the generation of chains of secondary requests it is possible that more than one response is sent to the client. The arrival of the last response of a series of responses is the completion of this request.

4.3 Clients (Proxies)

The *Clients* are represented by simple CSL nodes that generate primary requests. One of these nodes can represent an arbitrary number of real world clients by adjusting the interarrive time for the request generation. As responses return to clients, the latter collect data for future calculation of statistics. Clients also handle transactions. Messages, generated at a client, are routed to servers that are able to provide the requested services.

4.4 Servers

Servers are simple CSL nodes that host services. Any request, arriving to a server, carries information about the service that it is requesting. If this server hosts such a service and the server has enough available resources to render this service, it executes the service during some *service time* that depends on the relative service time assigned for this service and the “speed” of the server (its node delay).

It may happen then that a server issues secondary requests. These requests memorize the server that issued them and are routed to servers that can provide the correspondent services.

If secondary requests are not generated, the server forms a response that is routed back either directly to the client that originated the primary request or to the server that issued the last secondary request.

If a response arrives to a server as to an intermediate destination, it is forwarded either to the next server that is in the message’s list of servers that generated secondary requests or to the client, if this list is empty.

4.5 Clusters

A *Cluster* is an aggregate CSL node that consists of servers and/or clusters of servers. Services assigned to servers and subclusters of a cluster are considered to be assigned to this cluster. Each cluster can have a *Partition* that distributes services assigned to it among its servers or subclusters. This partition is static and does not change during a model run. The cluster can also balance the traffic of messages-requests to its subclusters and servers, hence to balance the service load at these subclusters and servers. For this purpose it monitors the current load to and from of all its services.

4.6 Transactions

A *Transaction* represents a group of messages (requests and responses), which serve to fulfill some user's task. The transactions are generated with some interarrival time and each transaction initiates a series of request messages. Some termination condition defines when a transaction is completed. To prevent overcrowding at services, an *admission control* for transactions may be used in servers. The class *AdmissionControl* defines admission conditions and what to do with the transactions that are not granted admission to services.

5 SERVICE PARTITION

An important factor of the traffic efficiency in a distributed service system is the partition of services among servers and clusters of servers.

Let us assume that there are *m* services that should be distributed over *n* server nodes. There exist several kinds of restrictions influencing this mapping:

1. Number of server nodes per service: A service can only reside on one server or a service can be distributed (replicated) over several servers.
2. Number of services per server node: Each server hosts exactly one service (If services cannot be replicated over several servers then $m = n$.) *or* each server can host a maximal number of x services (with $x \le m$).
3. Restriction on service placement: Some service has to be mapped on a certain server or some service has to be offered by a certain cluster.

Figure 4 shows an example that could be described by a user through input files.

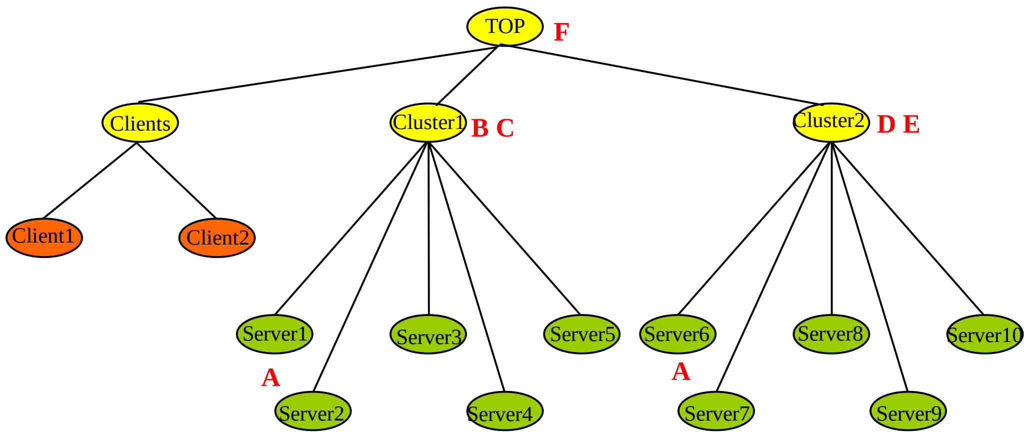


Figure 4

The topology consists of some clients and two server clusters with five servers each. The services of this example are named *A, B, C, D, E* and *F*. Each server can host only one service, but services can be replicated over several nodes. Service *A* must reside on *Server1* and *Server6*. Services *B* and *C* must be assigned to servers in *Cluster1* while service *D* and *E* are restricted to *Cluster2*. Service *F* can be mapped on any server.

The class *Partition* defines how services are mapped on servers respecting the given restrictions. In the example, the class Partition could create a random mapping of the services *A, B, C, D, E* and *F* on the ten servers. A possible result is shown in Figure 5.

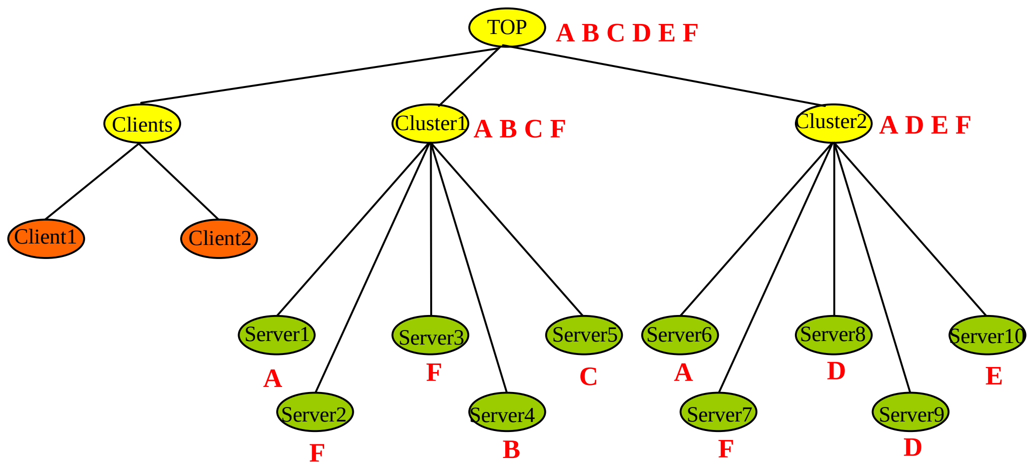


Figure 5

After the actual mapping is made, each cluster updates its list of services, which consists of all services offered by servers in this cluster.

Of course, a random partition of services is not very helpful and should maybe only serve as a default partition behavior. More sophisticated partition schemes could be for example:

- **Load distribution:** Services should be distributed among servers according to their expected load. A service *A*, which will cause twice as much load as a service *B*, should be mapped on twice as many servers as *B*. If several services per server are allowed then they should not be distributed in such a way that some servers are overloaded while others are idle most of the time.
- **Minimal communication costs:** Services that communicate with each other (via secondary requests) should be placed on servers close to each other, e.g. in the same cluster. This reduces the total bandwidth usage and traffic delays of secondary requests.
- **Optimal resource utilization:** The placement from services on servers could depend on some limited resources like main memory or disk space.

Other partition strategies can be imagined as well as combinations of them. Furthermore, partition schemes could vary between different clusters.

So far, one additional Partition class has been implemented. It considers the expected load situation and the communication cost between services.

Figure 6 shows another example for a partition of ten services S_1, S_2,.. , S_{10} on ten server nodes N_1, N_2, .. , N_{10} with the following restrictions:

- Each server can host exactly one service. Services cannot be replicated over several nodes.
- Clients create requests for service S_1. Each service S_i creates a secondary request for the service S_{i+1}.
- Nodes are connected in a chain where traffic costs between nodes increase proportionally to their distance. In this example, traffic costs just consider the number of hops between two servers.

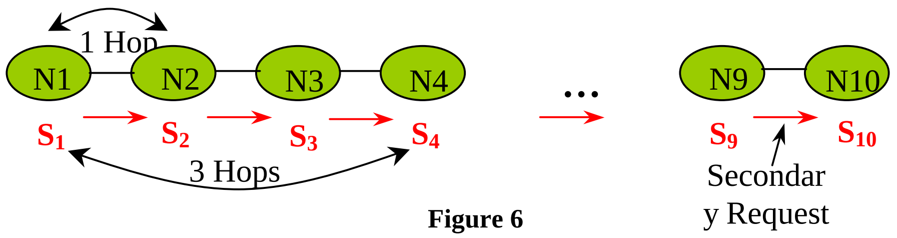


Figure 6

Obviously, the partition in Figure 6 is the optimal solution according to minimal communication costs in the system, because each secondary request has to travel only one hop to get to its destination node. However, automatically finding this solution in the set of more than 3.6 Million different mappings can be very time and resource consuming. Additionally, if the placement of services onto servers is less restricted (e.g. arbitrary number of services per server including replication of services) the number of possible mappings rises dramatically. Therefore, an exhaustive search is not a good idea, especially if one deals with systems of systems with hundreds or thousands of nodes. In the implemented partition class, it is tried to find a reasonable good partition by applying a genetic algorithm (GA) [3].

A genetic algorithm is an algorithm that incorporates aspects of natural selection or survival of the fittest. It maintains a population of chromosomes that evolve according to rules of selection, recombination and mutation. A chromosome represents a usually binary encoded solution to the problem being solved. In our context, it is one possible mapping of services onto server nodes. Figure 7 shows how the mapping of three services *A*, *B* and *C* on four server nodes can be binary encoded. Each bit represents a service mapping of one service to a certain node. The first bit (starting from left) codes that service *A* is mapped on *Server0*; the second bit shows that service *B* is not available on *Server0* etc.

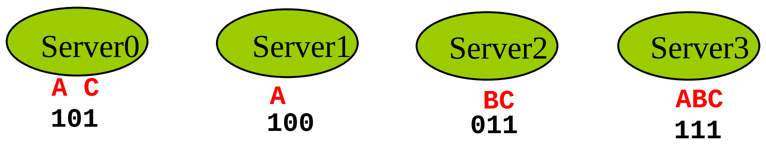


Figure 7

How good or bad a chromosome (which means one certain mapping) is can be evaluated by a cost function. With each new generation, the genetic algorithm builds new chromosomes whose cost values should be smaller, meaning that they represent a better service mapping. The cost value of a chromosome is calculated in the following way:

$$\cos t = \alpha \cdot loadCost + (1-\alpha) \cdot commCost$$

- *loadCost* is a value that represents how well the load is balanced among the servers. It is the standard deviation of the mean utilization[2] of all nodes. The smaller it is the better the load is balanced.
- *commCost* is a value that reflects the communication costs between server nodes caused by secondary requests. It is calculated with the following formula:

$$commCost = \frac{\sum_{i}^{n}\sum_{j}^{n} mess_{i,j} \cdot prox_{i,j}}{allMess \cdot \max Dist}$$

    - n is the number of server nodes.
    - $mess_{i,j}$ is the number of messages exchanged between server node i and j in a time unit.
    - $prox_{i,j}$ is the distance (in hops) between node i and j.
    - *allMess* is the total number of messages transferred between all nodes in a time unit.
    - *maxDist* is the maximum distance between two server nodes in the system.

    commCost is a value between zero and one. A value of one represents the worst-case scenario with a maximum number of messages transferred over the maximum distance. A value of zero means that there are no messages exchanged between servers.

- The weight factor α is used to prioritize one of the two cost values.

The genetic algorithm starts with a number of randomly selected chromosomes that fulfill the given restrictions. For example, if service *A* has to be mapped on *Server3* then the corresponding bit will be set in all chromosomes of the population. In the opposite case, if a cluster should only host the services *A, B, C* and not *D*, *E* then all bits representing *D* and *E* on the servers of this cluster will be unset in all chromosomes. These predefined bits of the chromosome will be referred to as **restricted bits**.

[2] A utilization of 1.0 means that a server is busy 100% of the time.

If *pop_size* is the number of chromosomes in a population then *pop_size*/2 randomly selected pairs of chromosomes produce new chromosomes for the next generation. A pair of chromosomes creates an offspring by joining segments chosen alternately from one of the two parents' chromosomes. Which segment is taken from which parent is defined by a randomly created crossover bit set. If the corresponding bit in the crossover bit set is one then it is taken from the first parent chromosome, otherwise from the second. Figure 8 shows an example of the crossing of two chromosomes. If the cost value of the offspring is better (lower) than the value of the worse parent, then the offspring replaces this parent in the population. This guaranties that the mean cost value of the population decreases from generation to generation resulting in better chromosomes.

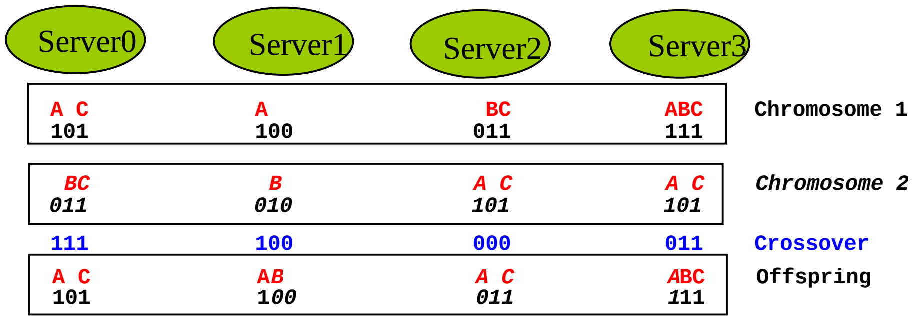


Figure 8

The second way of altering chromosomes is mutation. Mutations increase the search space of solutions by slightly changing chromosomes of the population. A mutation rate defines the probability with which each bit of the chromosome is flipped. Restricted bits cannot be changed by mutation.

Sometimes it can happen that a created offspring or a mutated chromosome breaks some basic rules of the underlying mapping. The following cases can occur:

- There could be a server without any services. This is corrected by mapping a random service on that empty server. This service must be one, which is allowed on this server.
- The number of services per server node can be restricted in a range from one to *m* services. If there are too many services on a server, some randomly chosen services will be deleted from this server. (This could lead to the next case.)
- There could be a service that is not mapped on any server. This service will be mapped on a randomly chosen server. This server must be allowed to host the service. If adding the service exceeds the maximal number of services per server, then the service is only added to this node if another service is found, which is also mapped on some other node and can therefore be replaced on this node.

The genetic algorithm can be tuned by adjusting the following parameter:

- Population size,

- Number of generations,
- Cross-over ratio (number of cross-over points in the cross-over bit set),
- Mutation ratio,
- Weight factor for cost function (to prioritize load balancing or minimal traffic).

After the genetic algorithm has finished, the best chromosome of the population will serve as the actual service partition of the model.

Returning to the example of figure 4, let us assume that there are three times more requests for service *B* than for *C* and that the services *D* and *E* call service *F*. The genetic partition algorithm could create a mapping like in figure 9 where service *B* is replicated over three servers and service *F* is only mapped into *Cluster2*.

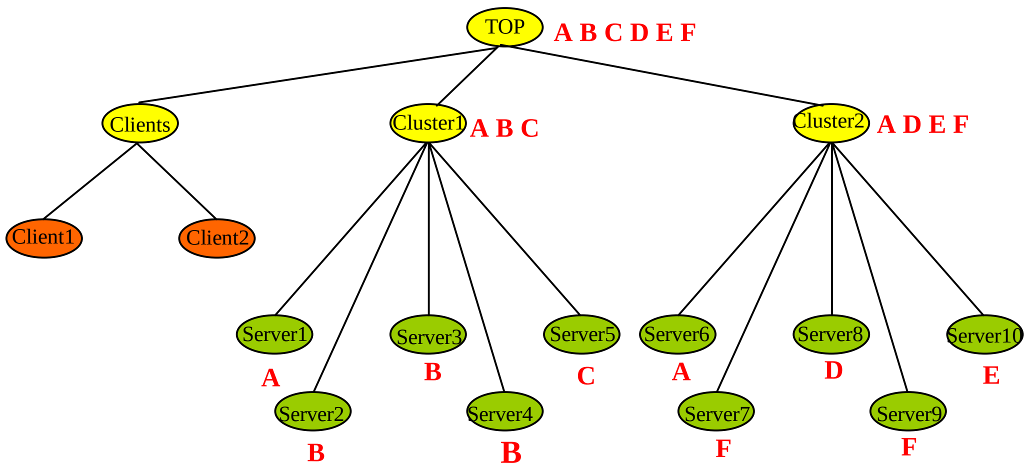


Figure 6

6 LOAD BALANCING

Load balancing is usually applied when service requests should be distributed among several servers in a cluster. One common example is a load-balancing device, which is placed in front of a cluster of Web servers. The load balancer forwards incoming requests according to a certain policy, which could be, for example:

- a random distribution,
- a round-robin distribution,
- sending the request to the server with the least number of open TCP connections,

- sending the request to the server, which has the lowest load according to some metrics, received from the server, e.g. CPU and memory utilization or number of running processes.

Such load balancing policies can be expressed in a Systems of Servers model. A class *LoadBalancer* exists which enables a cluster node to distribute service requests among the servers in the cluster according to some chosen load balancing policy. The policy, which considers the number of open TCP connections, is modeled by counting the number of requests sent to a server without receiving responses yet. Information from a server node, which can be used to make a load balance decision, is, for example, the queue length of the waiting requests. Other load balancing policies can be easily applied by deriving a new class from *LoadBalancer* that implements the desired behavior.

7 CACHING

Modeling caches can be done at different abstraction levels:

1. A cache can be considered as a service that is characterized by its hit ratio *hr* and a certain delay *d*. A client's service request will be fulfilled by the cache with the probability *hr* and the delay *d*. With the probability (1-*hr*), the cache cannot satisfy the request itself and will create a secondary request to the origin server. Characteristics like cache size or cache replacement polices are not modeled. An example is shown in Figure 10. A number of Web clients are connected to a

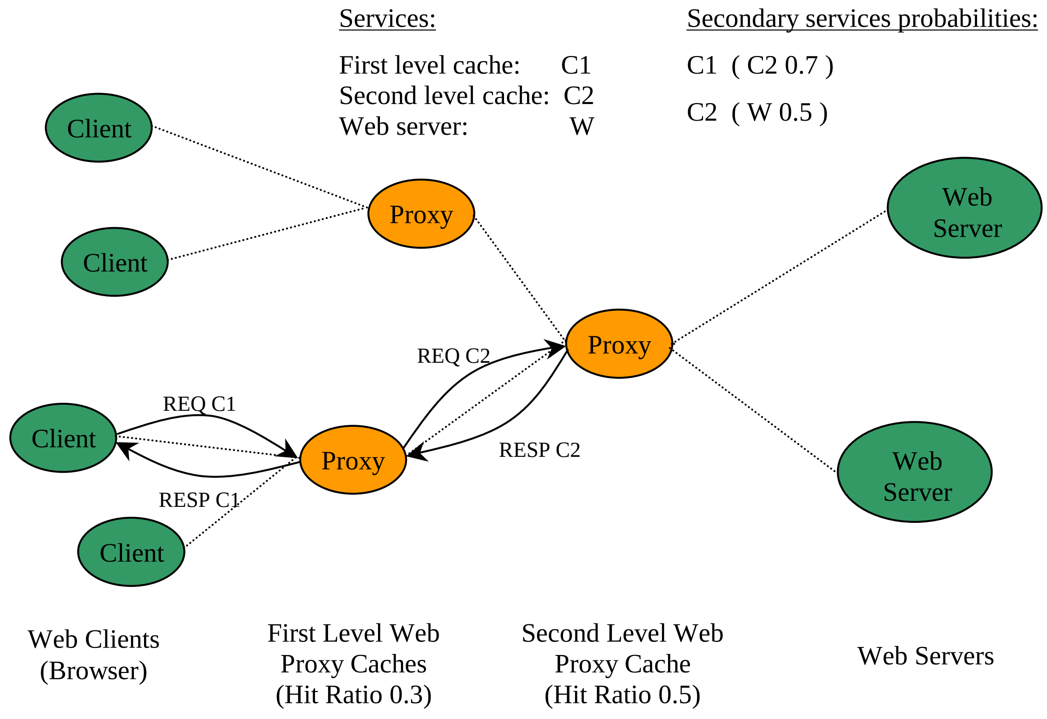


Figure 6

two level hierarchy of proxy caches. If a client's request cannot be served from the first level proxy, a secondary request is created to the second level proxy. Only if this one also cannot fulfill the request, the request is sent to a Web server. The miss ratios of the proxy caches are represented by the secondary request probabilities of the proxy cache service.

2. If the goal of a planned simulation is the observation of the cache behavior itself the above abstraction level is probably not detailed enough, especially if the cache's hit ratio is not known and should maybe obtained through the simulation. In this case a cache should be characterized by

   - a set of distinguishable objects, which can be cached,
   - the cache size,
   - a replacement policy if the cache is full and new arriving objects should be cached.

   This caching behavior has not been implemented in the Systems of Servers domain itself, but in a new domain library, which is derived from Systems of Servers. It is an example how Systems of Servers can be easily extended to model specific system behavior.

   A service request now includes an identifier (number) that allows distinguishing cacheable objects. The behavior of a server changes in the following way:

     - The creation of secondary requests now only depends on the fact whether the requested object is in the cache. The probability values from the service matrix used before are ignored in this case.
     - If a service request arrives at a server, it looks up whether the object with the given identifier is cached.
     - If it is cached, a response is sent back to the origin of this request and the cache is updated according to the replacement policy, e.g. changing priorities or the order of objects that will be replaced next.
     - Otherwise, a secondary service request will be created and sent to the next server according to the service matrix. If the response from this server returns the object will be cached, and a response will be sent back to the origin of the request. While caching this object, other objects may be removed from the cache according to the cache's replacement policy.

   Several classes of caches were implemented differing in their replacement policies, e.g. LFU, LRU, SIZE and GDS (Greedy Dual Size).

8 CONCLUSION

Systems of Servers is focused on the quantitative analysis and optimization of global system solutions. These solutions primarily influence the performance and cost of large distributed service systems. The library has been successfully used to validate global architectural solutions in enterprise computing infrastructures, multi-tier electronic delivery systems, and open distributed E-services.

Communicating Structures Library (CSL), Systems of Servers and specialized libraries built on the top of them, form *System Factory* [4], an open modeling environment for integration and customization of distributed system.

9 REFERENCES

1. Kotov,V.E. Communicating Structures for Modeling Large-Scale Systems. In Proceedings of the 1998 Winter Simulation Conference, Washington, D.C., ed. D.J.Medeiros, E.F.Watson, J.S.Carson, and M.S.Manivannan, 1998, pp.1571-1578.

2. Kotov,V.E., Rokicki,T.M., and CherkasovaL.A. CSL: Communicating Structures Library for System Modeling and Analysis. HP Labs Technical Report HPL-98-118, Palo Alto, CA, 1998.

3. Haupt,R.L. andHaupt, S.E. Practical Genetic Algorithms. John Wiley & Sons, 1998, 177 p.

4. Kotov,V.E. System Factory: Integrating Tools for System Integration. HP Labs Technical Report HPL-99-118, Palo Alto, CA, 1999.