Symphony: Managing Virtual Servers in the Global Village*
(White Paper – Preliminary Version)

Roy Friedman  Eli Biham  Ayal Itzkovitz  Assaf Schuster
Department of Computer Science
The Technion
Haifa 32000
Israel
{roy,biham,ayali,assaf}@cs.technion.ac.il

June 25, 1998

Abstract

A virtual server is a server whose location in an internet is virtual; it may move from one physical site to another, and it may span a dynamically changing number of physical sites. In particular, during periods of high load, it may grow to new machines, while in other times it may shrink into a single host, and may even allow other virtual servers to run on the same host. This paper describes the design and architecture of Symphony, a management infrastructure for executing virtual servers in internet settings. This design is based on combining CORBA technology with group communication capabilities, for added reliability and fault tolerance.

*Symphony is part of the “Virtual Servers – Infrastructure for Dynamic Access to Global Information” project, supported by the Israeli Ministry of Science.
1 Introduction

The emergence of internets, and in particular the Internet, created a potential for (literally) global resource sharing. If we examine the total load on all computing servers in the world at any given moment, we may see the following phenomena: While the load on a given computer may increase and decrease throughout the day, the overall load on all computers remains more or less constant. Moreover, at times when some servers are overloaded to the point of crushing, other machines are sitting idle.

To solve this problem, we propose the notion of virtual servers. A virtual server is a server whose identity is not bound to a fixed physical computer, but rather is changing and evolving with time and in reaction to the load on this server. Thus, a virtual server may move from one location in the network to another, and the number of physical computers on which it resides may change dynamically.

Consider, for example, an e-commerce application that shifts its location from the US to Japan when night falls on Los Angeles, and then from Japan to Israel when day breaks in Tel-Aviv. In this example, being able to shift the location of the server guarantees that at any given moment, the server is located in proximity to its current clients. Another example is a large ray tracing application running on a cluster of PCs in the Technion at night, using a distributed shared memory system like Millipede [12, 13]. This application suddenly notices that there are idle workstations at the Hebrew University in Jerusalem, and then decides to shift some of its computing tasks there, in order to enjoy higher parallelism. Similarly, Web based news sites become overloaded when an important event, such as elections or an important chess game, take place. These Web sites need to be able to grow, perhaps even to distant hosts, in order to handle the load.

In this work we describe the internal architecture of Symphony, an infrastructure for managing and executing virtual servers in internet settings. Symphony’s architecture is based on the fundamental principles underlying CORBA, and provide many of the standard services defined in CORBA [2]. In particular, it supports a distributed object oriented programming model, in which service interface definitions can be inherited and components of the system are interoperable. However, it is not a “pure CORBA” design, in the sense that services and applications (virtual servers) are replicated and can communicate internally in a non-CORBA compliant fashion. Also, we have taken liberty to extend the basic services of CORBA to fit our needs, as discussed later in this paper.

This distinction is important since it enables us to integrate high-performance servers, such as distributed shared memory, for which CORBA’s performance is insufficient at best. Also, our main management services, as described later in this paper, are built using the Ensemble [9] group communication technology [4]; as of today, there are no scalable solutions for directly integrating group communication with CORBA, and this design allows us to build scalable management services without giving up the power of group communication.\footnote{Voyager\textsuperscript{T M} by ObjectSpace\textsuperscript{T M} includes the notion of a space\textsuperscript{T M}, which is similar to group communication. However, the semantics provided for spaces in Voyager\textsuperscript{T M} is weaker than the usual semantics provided by group communication systems. See Section 7 for further discussion on combining CORBA and group communication.}

A service based design has the benefits of scalability, interoperability, and extendibility. The design is scalable, since the location of a service in the system is virtual. A service can run anywhere from a single host to the entire set of machines, yet be accessible everywhere. Also, services present an abstraction of objects, where only their interface is known, so that different implementation of the same service can be replaced without having to recompile the system. Finally, it is always possible to add services, or modify existing ones, in order to extend the functionality of the system.
In summary, our design tries to enjoy the benefits of both worlds. We use group communication as the main building block for our management infrastructure, use a CORBA-oriented architecture, and allow replicated/distributed servers to communicate internally with their own communication stacks for high performance. (Of course, servers that wish to communicate internally using CORBA can do so, but we do not impose this on other servers.)

Finally, Symphony is an infrastructure technology whose purpose is to aid in the design and implementation of virtual servers. Clearly, any distributed application can be implemented directly on top of the raw networking support provided by any modern operating system. However, the same argument can be said in favor of using bare hardware rather than advanced operating system’s services. Yet, most people prefer to use high level abstractions to develop programs. We believe Symphony should be viewed in a similar way w.r.t. to developing virtual servers.

The rest of this paper is organized as follows: Section 2 describes the general architecture, and describes the main services we support and the interaction between them. Section 3 discusses security aspects of Symphony, while the scalability challenges of Symphony are presented in Section 4. Section 5 elaborate on Symphony’s services. We provide a few example applications in Section 6, compare our work with related projects in Section 7, and conclude with a discussion in Section 8.

2 Architectural Overview

Our architecture follows the CORBA approach of providing services to perform common tasks needed by applications and other services in the system. We arrange our services in several domains, as depicted in Figure 1. Services are logically organized in a hierarchical structure such that services in one domain tend to draw upon the functionality provided by services in lower or equal domains, but not in higher domains, as discussed below. Also, Symphony’s run-time system is located below all services alongside with the operating system.

Management services lie at the heart of our design. These services provide support for automatically replicating and migrating virtual servers based on the characteristics of these servers, the topology of the network, computers availability and so forth. The application can register with these services a description of its requirements and cost functions for locating “ideal” machines for it. The application is free to choose between two modes of operations: (a) The application can either allow the management services to automatically migrate and replicate itself based on the above descriptions (and restrictions), or (b) request non binding recommendations from the management services, but maintain the final word as to where and when its replicas should be (re)moved. This latter option is given for virtual servers that wish to retain control regarding the location of their replicas.

Management services draw upon the functionality provided by both informational services and code-handler services. The code-handler services do the actual task of starting an image of a code on a given machine upon demand, or evacuating machines under certain conditions. These services may receive instructions from either management services or applications, receive operational information from informational services, and utilize the functionality of the basic services as part of their normal operation.

Informational services, on the other hand, collect and report information about the system. This information includes the load on participating machines, the topology of the system, the given environment on each of the machines, and the location of various applications. The information gathered by informational services can either be reported to subscribers using registered up-calls, or given as a response to an explicit request.

Basic services are located at the bottom of the hierarchy. These services provide the basic
Communicating with Replicated Services

Most of the services we propose need to be replicated for efficiency and high-availability. Also, different services may need different levels of replication. Some services may require a replica on every host, e.g., the code-dispatcher, while for other services it may be sufficient to instantiate a single replica in each cluster or subnet.

This goal is realized by splitting each service into a local stub that has to be started on each host, and a global part that is replicated, as illustrated in Figure 3. The global part consists of several replicas that form a process group; these replicas communicate through a group communication system that also manages membership changes, i.e., discovery of both failures of existing replicas
and joins by new replicas.\textsuperscript{2} The local stub communicates directly with the application or service.

\textsuperscript{2}The Event Notification Service of CORBA can be viewed as a plausible alternative for implementing replicated services. Felber and Guerraoui explain in [7] why ENS is in fact unsuitable for this task.
entity on the local host. When contacted, the stub issues a multicast RPC request to the global part of the service, and is then responsible for collecting the replies and forwarding them to the local entity that initiated the request. In particular, the local stub can be asked to deliver the first reply that arrives from any replica, deliver all replies, or deliver the most common reply. Also, invocations on the local stub may be either synchronous, deferred synchronous, or asynchronous [16].

2.2 Programming Model

Developers of virtual servers in Symphony are not obliged to use a specific programming model. Instead, Symphony’s services are made available to virtual servers applications to be used at will. Nevertheless, we expect most developers to use one of the following programming models:

“Raw” Group Communication: Symphony is based on the Ensemble group communication system [9]. Thus, virtual servers developed for Symphony can use the “raw” interface provided by Ensemble. This interface supports creating and joining groups, sending and receiving messages to members of the group, and handling view changes, i.e., changes in the configuration of the system.

Object Oriented Group Communication: Ensemble comes with an object oriented programming interface, called Maestro [19], that can be accessed either from C++ or Java. Maestro supports similar methods as the “raw” interface of Ensemble, but present them in an object-oriented manner, and adds better support for state transfer.

CORBA: CORBA is a common programming paradigm for developing distributed applications, and in particular Symphony is based on CORBA. We expect that it to be one of the common development paradigms for virtual servers.

Java RMI: The Java RMI mechanism offers similar distributed programming capabilities as CORBA, with a somewhat simpler programming environment. Java RMI, however, does not allow asynchronous and one-way invocations, and is limited in its capability to access legacy systems. However, for new applications that need not interact with older systems, and in which performance may not be the most important aspect, Java is likely to be a popular programming paradigm.

2.3 State Transfer

Note that when moving a virtual server from one host to another, it is important to preserve the state of that server on the new host. However, the level at which the state of an application needs to be preserved depends on the application. In many cases only a certain part of the application level state needs to be recovered, and this part of the state can be easily encapsulated. In these cases, the application can use the persistence service to dump the important part of the state to disk, or to another computer on the network, so it can be retrieved later.

Similarly, if due to a failure, one replica takes over the functionality of another replica, the latter replica should continue with the same state as was last held by the failing replica. In Symphony the application can either maintain consistency explicitly, possibly by using a group communication toolkit or the transaction service, or register replicated objects with the replication service, automatically takes care of maintaining a consistent state among all replicas, as discussed in Section 5.4.

On the other hand, if no assumptions are made about the application, the entire process state needs to be recovered, including all registers, open files, and memory image (both stack and heap).
This is a complicated operation, especially if the architectures of the host from which the state is taken differs from the architecture of the new host in terms of byte ordering or registers. At the initial phase, Symphony only supports complete state migration between similar architectures, and is part of the responsibility of the code-evacuator service.

3 Security in Symphony

A major concern when allowing servers to communicate over internets and allowing code to migrate from one place to another is security. Our approach to security is based on a trust model that is enforced by the configuration and security services. In this section we first describe the trust model, and later elaborate on how messages and code are secured.

3.1 Trust Model

A trust model, as suggested by its name, defines for each entity the kind of trust it has in other entities, and how this trust is verified. In Symphony, an entity can be a host, a message, or an image of a code. Each entity $e$ holds a data structure that contains the following fields:

1. list of entities to whom $e$ is willing to send messages.
2. list of entities from whom $e$ is willing to accept messages.
3. list of entities to whom $e$ is willing to send code.
4. list of entities from whom $e$ is willing to accept code.
5. list of entities to whom $e$ is willing to migrate/replicate.
6. list of entities to whom $e$ is willing to provide services.
7. list of entities with whom $e$ is willing to share resources.

Each of these categories is defined by specifying an access control list, and can be refined using the following subcriteria:

a. type (of communication/service/code).
b. authentication scheme.
c. limitations on the way the communication was carried, e.g., has to go through a firewall.
d. other conditions for the interaction.
e. recommendations in case multiple options occur.

Thus, it is possible to specify that host $A$ is willing to accept any code written by Assaf, if it is authenticated using the El Gamal scheme, and it passed through a given firewall. $A$ may also be willing to accept any Java applet that was generated at the Technion, and is signed with an MD5 hash function and a known secret key. Similarly, it is possible to specify that a replica of “Your-Fastest-Web-Search-Engine” never runs on the same machine as a replica of “The-Most-Complete-Web-Search-Engine”, while some server $A$ may require encryption of queries, but is willing to receive authenticated, but not encrypted, replies.
The trust model we have just described is not only used by virtual servers, but also by Symphony’s internal services. In particular, some of the management and informational service may not accept certain types of data from untrusted entities. This is important in order to prevent malicious parties from creating havoc in the system. For example, if the host recommender recommends bad hosts, it may cause the system to migrate code to the wrong nodes, which will degrade the performance of the system, and make it unusable.

3.2 Message Security

There are two aspects of securing messages: authentication and encryption. The former incurs relatively low overhead, but the latter is quite expensive. Similarly, most forms of authentication are publicly available, and can be distributed freely. On the other hand, encryption is usually protected by patents, and its distribution is usually limited by export regulations.

Symphony allows virtual servers to choose which of the two they wish to employ, in order to obey the trust model. As with other concerns, authentication and encryption are supported by appropriate services. Moreover, as some applications may prefer to do authentication and encryption themselves, Symphony provides key management services; these services distributed keys among replicas of subscribed virtual servers, revoke expired keys, and initiate key replacement according to given specifications.

3.3 Securing Imported Code Execution

In recent years there has been a large body of research on how to securely execute imported code. In Symphony, an exported code is accompanied with a descriptive object, which defines the restrictions the code is willing to run under, as well as the runtime environments it requires. Also, part of the information gathered by the informational services is the policies employed at each of the sites on the network towards such code. In particular, a code can be authenticated as coming from Assaf at the Technion, and some hosts in Jerusalem may be configured to allow full execution rights to Assaf’s code, but restrict all other code to a Java-like sand-box model.

When sending code for remote execution, the code-dispatcher service consults with the appropriate information services regarding the possibility of placing the given code on the requested machine. The code is then executed on the remote site only if the site’s policy allow the code to run there.

Moreover, the descriptive object accompanying the code includes the maximum amount of resources the process needs. Before starting a code image, the code-dispatcher verifies that the hosting machine is willing to provide that much resources to the imported code. These resources include memory consumption, disk space, cpu time, communication, and processes. If the answer is no, it will not be started on the given machine. If the answer is yes, it will be started, but the amount of resources it consumes will be monitored periodically; if they are exceeded, the process(es) will be stopped.

4 The Scalability and Extensibility Challenge

Symphony is designed to coordinate a large number of virtual servers running on a large number of nodes in a heterogeneous network, both in the types of computing nodes and in the network topology, and spanning large geographical areas. Coordinating actions in such a complex and large
environment is not an easy task. Also, being able to support a wide variety of virtual servers, which
typically have very little in common, is not a trivial task.

It is our experience that group communication technology is a powerful building block for develop-
ing cluster management utilities. Group communication often shortens the development process of
such applications, and yield a much more solid code than code written in an ad hoc manner. For this reason, we have decided to base our core management services on group communication
technology, more specifically, on the Ensemble group-ware system [9]. However, existing group
communication systems are limited to coordinating a few dozens of processes, and become unstable
when operated across wide area networks.

A major challenge of this project is to extend the Ensemble group-ware with scalable protocols
that can meet our needs. A first step towards scalability was achieved with the addition of the PBC-
CAST [10] protocol to Ensemble. In this work we expect to develop even more scalable protocols,
and to use a hierarchy of groups, along the lines of [3] to be able to cover thousands (and perhaps
even tens of thousands) of machines. These results will be reported in a separate paper.

Also, our CORBA-oriented design and the choice of services are targeted towards maximum
extensibility. Our design allows to combine a large number of existing virtual servers, and to
efficiently develop new virtual servers all within the same framework. In particular, it is possible
to add new services to the system, and to extend the functionality of the existing services in a
transparent way, if the need arises.

5 Detailed Description of Services

5.1 Management Services

Most management services deal with replicating, migrating, and updating virtual servers. This is
done by registering a given code with the appropriate service, and indicating a policy for treating
that code. Also, as discussed in Section 3, when a code is registered, it is accompanied by a
descriptive object, which describes the minimal requirements of the code, as well as identification
information about it. This includes the minimal set of permissions needed by it (read-files, write-
files, execute-files), minimal and maximal system resources (CPU, memory, disk, communication),
the owner of the code with authentication information (digital signature).

It is also permissible to register a list of possible code images, each with its own list of descriptive
objects. In this case, an indication of how to choose among these images is also supplied. These
policies include first-fit, i.e., run the first image that can be executed on the given host, best-fit, i.e.,
run the best image among those that can run there using a given evaluation function, all-fit, i.e.,
run all images only if all of them can run, or any-fit, i.e., run every image that can run on the host.

5.1.1 Code-replicator

The code-replicator can automatically replicate a virtual server on idle machines. When the appli-
cation is first started, it registers with the code-replicator a replication policy for the application.
For example, if an application incurs very little internal communication, any idle machine might be
a good candidate. On the other hand, if there is extensive internal communication, it makes sense
to start new replicas only on hosts that have sufficient bandwidth. This service chooses machines

\[\text{PBCAST was developed as part of the Ensemble project at Cornell University, in an unrelated effort to this project.}\]
based on recommendations provided by the host-recommender, and utilizes the code-dispatcher to start the code on a chosen machine. Both services are discussed below.

5.1.2 Code-migrator

The code-migrator is similar to the code replicator, except that it migrates the application rather than simply replicate it. In particular, this means that a code may need to be evacuated from one machine and be started on another. Typical reasons for migrating a replica of a server to another machines include, e.g., keeping the virtual server close to its clients, or the discovery of a more powerful idle machine.

5.1.3 Host-Recommender

The host-recommender tries to find ideal machines for registered applications. This is done by applying a cost function that is provided alongside each registered application to the current state of the system, as reported by the informational services. Hosts that come up are then reported by invoking an appropriate upcall.

5.1.4 Configuration Manager

This service allows hosts to register their policy regarding execution of imported code. This includes which code can run with which access rights, what authentication methods are accepted, the operating system and hardware architecture used, etc. The information is then registered with the appropriate information services, and can be used by other services and applications, e.g., by the code-dispatcher to decide if a submitted code can be started on a given host.

5.1.5 Monitor

The monitor service monitors the execution on any given hosts, in order to enforce execution policies, prevent anomalies, and react to them if they occur. In particular, the monitor service monitors the resource utilization of imported code images, and if a process running such code violates its resource limitations, then this process is stopped, and in some circumstances, the corresponding code image is evacuated. Also, if the load on a given machine grows too much, the monitor may decide to stop and perhaps even evacuate processes running imported code, even if none of these processes consumes more resource than it is allowed. This ensures that processes that are native to the local host are not endangered by imported ones. Also, this serves as a safety guard against erroneous decisions by other services, that caused one host to become overloaded.

5.2 Informational Services

Informational servers are somewhat of a special case in Symphony. Like all replicated services, these services have a local stub and a global part. However, in all informational services the local stub can also initiate communication with the global part on its own, rather than simply handle remote method invocations and replies. That is, the local stub periodically, and in some occasions as a result of significant changes, report the local state of the system, be it load, environment, etc., to the global part without an explicit request from any other local entity.

Moreover, the local stub can be polled locally as to the state of the local machine, in which case it does not communicate with the global part, and can periodically update local entities (applications and other services) about changes in the local host. Due to the vast amount of the
collected information, each informational service has methods for requesting digested information for specific hosts and/or specific IP domains, as well as for the entire systems.

We have defined the following set of informational services to be supported in Symphony:

5.2.1 Load

Monitors the load of the system. This includes all standard computing resources such as CPU utilization, memory consumption, disk accesses, networking, etc.

5.2.2 Topology

This service tries to figure out the topology of the system; which machines are alive, how they are connected, and the quality of communication between various nodes.

5.2.3 Environment

This service keeps track of the architecture of the local host, operating system, supported run-time environments, access control list restrictions, etc.

5.2.4 Application Locator

The application locator monitors the applications located on any given host in the system. In particular, it has methods for retrieving the list of applications that are running on a given host (or set of hosts), and which hosts run a given application (or set of applications).

5.3 Code Handling Services

5.3.1 Code-Dispatcher

The code-dispatcher is responsible for actually starting an image of a code on a given host. This is done by invoking the dispatch method on the code-dispatcher service. Aside from the executable image, or applet, this method accepts a descriptive object describing the requirements of the code. These include the run-time environment it requires, the operating system, and the hardware architecture. Other parameters include memory consumption, disk space, and CPU units. Of course, for each of these parameters it is possible to indicate don\text{-}not\text{-}care if these are unimportant. In addition to the above parameters, the descriptive object includes a list of restrictions under which the code can execute, as discussed in Section 3.

After receiving the code and its descriptive object, the code-dispatcher checks with the appropriate information services whether the requirements of the descriptive object hold on the requested machine. If they do, the code-dispatcher writes the image of the executable/applet on the remote machine, and starts executing it with the requested run-time.

5.3.2 Code-evacuator

The code evacuator, as suggested by its name, is responsible for evacuating code from a given machine. It is invoked with an evacuate method. The evacuated code image, including registers, stacks, and heap, and any output it has produced, both to files and standard output can either be discarded, or returned to a given host, as discussed in Section 2.3.
5.3.3 Code-updater

The code-updater is provided as a mean for updating replicas of a given service on-the-fly. Typically, the code-updater shuts down a replica, updates the code image, and then starts the new code image on the same host. By repeating this process to all replicas, one by one, the entire set is being upgraded, while keeping (at least one replica of) the service on-line at all times.

5.3.4 Code-mover

The code-mover supports the move method for moving a given code from on host to another. This method employs both the code-evacuator and the code-dispatcher to actually move the code. The code-mover is different from the code-migrator in that it does not take decisions by itself, but rather simply executes explicit instructions regarding which code should be moved from which machine and to where.

5.4 Basic Services

In this section we discuss the basic services provided by Symphony. These services largely follow the CORBA standard, although we have extended their functionality beyond the requirements of CORBA, and added a few additional services, to fit our needs.

5.4.1 Naming

As in any CORBA system, objects, including services and applications, have names that identify them in the system. However, these names serve as symbolic links to objects, since they do not define where the object is located and how to access it. The naming service provides this capability. That is, in order to access an object for the first time, the equivalent of a UNIX fopen command must be issued. This command invokes the name-resolution method on the naming service, which returns a handle to the object. The handle to the object defines how the object should be accessed, how it can be located, and its current location.

Following this, whenever object $A$ invokes a method on object $B$, $A$ uses the handle to identify the requested objects. This invocation tries to invoke the requested method on $B$ by using the currently available information about $B$. If this attempt fails, the client’s server stub checks with the naming service for the current location of the $B$. If at least one copy of $B$ is still active and connected to the system, the naming service returns a new binding for $B$ that matches its current location.

This protocol also defines an interaction between the naming service and the application-locator service, since the naming service may need to consult with the application-locator as to the current location of an object.

5.4.2 Security

The security service supports signing, encrypting, authenticating and decrypting of messages, and methods for key management among a group of applications or services. These are used to enforce the trust model, as discussed in Section 3.
5.4.3 Transactions
According to the CORBA standard, the transactional service supports distributed transactions using the 2-phase commit protocol. We intend to extended this definition to support the more sophisticated 3-phase commit protocol.

5.4.4 Persistence
Objects can be registered with the persistence service if they need to be persistent. For example, according to the CORBA standard, the persistence service can flush (create a check point) an object to disk either periodically, or when certain events occur. We intend to extend this definition and allow the persistence service to flush an object to another host on the network as well as to the local disk. Recall that this can be used to migrate a virtual server as discussed in Section 2.3.

5.4.5 Replication
Replicated servers can register replicated objects with the replication service. In this case, the replication service takes care of maintaining consistency for these objects across all replicas. Symphony’s replication service supports three types of consistency: strong consistency, also known as sequential consistency, weak consistency, and dirty consistency. This service does not appear in the CORBA standard, although we view it as extremely useful in developing replicated applications.

5.4.6 Event Notification
The event notification service allows objects to register and multicast messages/event to interest groups. Messages are then delivered to all interested parties according to their topic. This mode of communication decouples senders from receivers, since a sender never knows who are the listeners.

5.4.7 Visualization Service
The visualization service takes the information from the informational services regarding the current configuration of the system, and visualize it for the system administrator for monitoring and maintenance. In particular it can draw a topological graph of the available machines, their current load, the servers running on each of them and their status. Due to the large scale of the system, this service needs to allow zooming in and out on various locations in the system.

6 Example Applications
In this section we present two example applications of Symphony, namely, a MicroMint and exhaustive search. Other applications under development using Symphony include a distributed Web server, a video-on-demand server, and a distributed video conferencing server.

6.1 MicroMint
MicroMint is a scheme for producing digital money for micropayments, known as microcash, devised by Rivest and Shamir [18]. In a MicroMint setting, the mint produces digital coins that expire within a relatively short period of time, e.g., within a month. At the beginning of each period, new currency becomes available for general usage, and is distributed to the public through intermediate branches, e.g., ATMs. Each digital coin can then be used once, and only once, in order to pay to
a store. The store returns the money to the mint and is credited accordingly. At the end of each period, the unused digital coins need to be returned to the mint for appropriate credit.

The mint verifies that each returned coin is legal, i.e., was not forged or used, before crediting the entity that brought the coin. Also, authentication schemes are used to detect violators that attempt to use a given coin more than once.

One of the advantages of the MicroMint scheme for micro payments is that each entity can easily verify that a coin is legal, although producing coins is very time/resource consuming, which makes forgery non-economic. The key to this is the structure of a coin; each coin is composed of two large numbers that are mapped by a well known hash function to the same hashed value.

Generating coins involves computing the hash value of many numbers, and trying to find pairs of numbers whose hash collide. By using \( n \) bit hash values, the first collision is discovered after roughly \( 2^{n/2} \) values have been computed, and from that point on, the number of collisions grows quadratically with the number of computed values.

Note that while calculating each hash is fairly trivial, computing \( 2^{n/2} \) hash functions is a time/resource consuming activity for large \( n \)'s (e.g., \( n > 80 \)). Also, the exact hash function is replaced with each period, and is kept secret until the beginning of the period. Thus, a good choice of a period can prevent even malicious resourceful parties, e.g., a hostile government, from generating coins before the end of the schedule.

There are certain aspects of the MicroMint scheme that makes it a good candidate for distribution, and in particular for being implemented using Symphony. Since generating a coin and verifying a returned coins are simple tasks, and these tasks are also independent of one another and of similar activities for other coins, they can be easily distributed. Also, the task of computing hash values, sorting the computed hash values in order to find duplicates, and verifying returned coins reach their activity peaks in different parts of the period. For example, at the end of a period, many coins are returned, while collisions can be found after roughly half the period. This behavior is mapped very well into Symphony’s support for growing and shrinking applications. Finally, as we explain below, MicroMint’s security aspects map very well into Symphony’s security model.

6.1.1 Implementing a MicroMint with Symphony

In the discussion below, we refer to seven functional entities: mint controller, value-generators, matchers, verifiers, distributors, clients, and stores, as depicted in Figure 4. Note that this illustration also indicate the security aspects that are discussed in Section 6.1.2. Figure 5 sketches a qualitative time-line for the proposed mode of operation in the MicroMint, while Figure 6 includes code extracts that show how different components of the mint interact with each other.

The mint itself is composed of a mint controller, value-generators, matchers, and verifiers in the following fashion: The mint controller informs other components of the mint at the beginning of each period about the new hash function for the next period, and informs the rest of the world about the hash function for the period which has just started. The generators compute hash values and send them in batches to the matchers, who try to find duplicates. Also, distributors contact matchers in order to obtain new coins for distribution to clients. Clients then pass coins to stores in order to pay for goods and/or services. Finally, stores and clients return their coins to verifiers, who inform the mint controller on any attempted forgery or dual usage of the same coin.

All components of the mint are distributed for fault-tolerance and efficiency. The mint controller is replicated, but has a small, fixed set of hosts it is constantly running on. On the other hand, value-generators, matchers, and verifiers can be located on any host with the appropriate security attributes, and in particular, replicate and vacate themselves as the load changes for maximum
efficiency. All hosts belonging to the mint are protected by a firewall from the rest of the network.

Distributors can also replicate themselves and migrate in case there is a sudden high demand for their services in some part of the system. Each distributor must also be protected by a firewall from the rest of the world.

Finally, stores and clients are outside the scope of symphony, but they communicate with certain parts of the mint as described above. All communication inside the mint, between the mint and the distributors, and between clients/stores and the verifiers must be authenticated and encrypted. It is advisable to do the same for communication between clients and stores, however, as they are outside the scope of the system, it is up to them to decide.

6.1.2 The Trust Model

We define four security levels with regards to the MicroMint: The mint controllers are assigned the highest level of security, followed by the value-generators, matchers and verifiers. The third level is given to distributors, while clients and stores share the lowest level of security.

The task of the mint controller can be held only by a limited number of known hosts. Thus, object descriptor for a mint control would specify that it can run only on one of these machines, and that it is willing to communicate only with other mint controllers, generators, matchers, and verifiers. This communication must be encrypted and authenticated.

Communication with other mint controllers is permitted only if they are in the same LAN and belong to the predefined list of mint controllers. All other communication inside the mint is permitted either if it takes place within the same LAN, or if it passes through the firewall. In any case, all messages must be encrypted and authenticated.

Value-generators, matchers, and verifiers can be replicated to any host that can be trusted by the mint controller, and are willing to communicate with the mint controller. In addition, each replica of a value-generator is willing to send messages only to matchers. A matcher is willing
(a) Mint controller is started on designated machines using the configuration service

(b) Mint controller registers itself with the naming service

(c) Mint controller registers with the code replicator the code image and descriptive object for value-generators, matchers, verifiers, and distributors.

(d) The code replicator asks the host recommender to recommend hosts for value-generators, matchers, verifiers, and distributors according to their descriptive objects

(e) The host recommender, based on information it gets from informational services recommends hosts to the code replicator

(f) The code replicator asks the code-dispatcher to start an image of the appropriate code on the corresponding hosts

(g) The code-dispatcher checks if the requested hosts are indeed valid based on the code’s descriptive object, information gathered from informational services, and the configuration service on the targeted host. If the answer is yes, the code is started there; otherwise, an error code is returned to the code replicator.

Figure 5: Starting and replicating MicroMint’s components. Note that Step (e) repeats itself whenever new hosts are found.
Mint Controller

... LastMonthKey = NewKey;
NewKey = GenerateNewKey();
refGenerators = Name.bind(“value-generators”);
refMatchers = Name.bind(“matchers”);
refVerifiers = Name.bind(“verifiers”);
refDistributors = Name.bind(“distributors”);
refGenerators→NewMonth(NewKey);
refMatchers→NewMonth(NewKey);
refVerifiers→NewMonth(NewKey);
refDistributors→NewMonth(LastMonthKey);
...

Value-generator

... for Current = BeginRange to EndRange do
    NewVal = ApplyHash(Key, Current);
    refMatchers→NewValue(NewVal);
od
...

Distributor

... AddCurrency(refGenerators→GetCurrency);
...

Figure 6: Sample interaction between MicroMint’s entities.

to communicate microcash on demand also to distributors, and verifiers are willing to receive microcash for verification only from clients and stores. Here again, all communication must be authenticated and encrypted, and has to go through a firewall if it is not within the same LAN. In particular, communication with distributors, clients and stores need to go through a firewall as well.

Distributors, on the other hand, are only willing to request microcash from matchers, and to distribute it to clients on demand. Communication between distributors and clients has to go through a firewall as well.

Finally, each MicroMint entity imposes the restriction that it is not willing to reside on the same host as entities that have lower security level. Thus, if a host is occupied by a distributor, it may no longer be used by a value-generator, etc. However, generators and matchers are allowed to run on the same host.
6.2 Exhaustive Search

Exhaustive search is an application that can be distributed/parallelized in a trivial way. The efforts to break the RSA data security DES/RC5 challenges, e.g., the RC5–64 and DES-II-2 projects [1], by splitting the search space among thousands of hosts is a good example of using distribution for conducting exhaustive search.

Implementing such a scheme in Symphony is trivial. One only needs to create a couple of simple applications. The first application, a searcher, receives a search space and an element to find, and then exhaustively searches it until either it finds the missing element, or until the whole subspace is completely searched. This code can be replicated to any idle machine in the network.

The second part of the code consists of the coordinator, which assigns search spaces to searchers. Also, each searcher that either found the missing element or finished its subspace, contacts the coordinator. If the missing element is found, the coordinator can either send an abort message to all searchers, or stop assigning new search spaces to them. Otherwise, each searcher that finished its subspace is assigned a new one. The coordinator can be replicated for reliability among two or three designated machines.

A single image of the searcher’s code can be registered with the code-replicator, which takes care of replicating it until it receives a command from the coordinator to stop. All replicated coordinators can be implemented using an object group, so they are kept synchronized regarding the state of the search. Also, the application locator and naming service can be used for transparent communication between the searchers and the coordinator. Hence, the system needs to be started once, and from that point on, no human interference/coordination is required.

7 Related Work

Amoeba is one of the first distributed operating systems [15]. Amoeba is also based on a group communication system, although the group communication system used by Amoeba was developed as part of that project. In important difference between Amoeba and Symphony is that Amoeba is a full operating system, while Symphony is not. In particular, Amoeba deals with process scheduling and Symphony does not. This is because the two system have different statement of mission. Amoeba was built right from the start to be a distributed operating system, while Symphony is intended to allow users to run virtual servers in a connected world, which is composed of computers running existing operating systems. Another significant difference is the service-based architecture of Symphony, which is different than the monolithic architecture of Amoeba.

Code images of virtual servers might be compared to mobile agents, as implemented in the TACOMA project [14]. However, agents in TACOMA are intended to move around freely in the Internet, without trusting anyone, and without being trusted by any hosting computer. Thus, a large part of the effort in TACOMA deals with protecting agents against malicious attacks, and protecting hosts against malicious agents. In Symphony, we use a trust model, which circumvents most of these problems, but is not suitable for agents. On the other hand, TACOMA does not handle issues like global load balancing and migrating code based on performance considerations. Of course, TACOMA could serve as a platform for developing a system like Symphony, but at its current state it does not provide this functionality.

Legion [8] is another object-oriented distributed system that provides an illusion of a single virtual machine composed of wide area collection of workstations, parallel computers, and fast massive multiprocessor machines. Class interfaces in Legion can be written in either CORBA’s IDL or MDL [8]. Unlike Symphony, in which an application can choose the set of service it wishes to
use, Legion requires all objects to conform to its model. Each object should inherit from a predefined LegionObject and each class should be also instantiated and inherited from a base class named LegionClass. This programming model (rules) enables the Legion runtime system to efficiently support the mobility of application’s objects, security in object invocation and heterogeneity. Legion deals with issues of scalability, security, fault-tolerance, and management of a large scale (sometimes called nation-wide) meta-computer. Note that, Legion address scalability by cloning objects, but does not take Symphony’s approach of replicated services based on group communication.

Hadas [11] is a programming environment and runtime infrastructure for composing distributed applications from independently developed and widely dispersed components. Working in Hadas, component developers create (or encapsulate pre-existing) components and publish them in the distributed component management system. In particular, Hadas components are dynamically adaptable, allowing autonomous maintenance and evolution without disrupting the application or other components. Hadas also introduced the notion of Ambassadors; an Ambassador is a representative of a component that gets deployed and is attached to a remote component upon successful negotiation between two sites. Implemented itself as a dynamically adaptable object, it can be tailored specifically to the foreign application and to the remote environment, and evolve on-the-fly to reflect changes that occurred after it has been deployed, both “at home” and at the hosting site.

Java is a programming language and a run-time system that supports an object oriented design. In particular, the RMI mechanism of Java allows to implement distributed object oriented systems. CORBA, on the other hand, is an international standard that defines a framework for designing and implementing distributed object oriented systems. Also, CORBA allows to mix objects developed in different programming languages, and defines standard services that are important for designing distributed systems. In particular, CORBA objects can be written in Java, and there are ORBs written in Java. Both technologies lack means for migrating and replicating virtual servers, and for global load balancing. Symphony employs both technologies in providing this functionality.

There are several known approaches for combining CORBA and group communication. Electra and Orbix+ISIS are examples of attempts to integrate a group communication system within the ORB[16]. The main disadvantage of this approach is that it requires changes in the code of the existing ORB. Also, these specific examples had serious scalability problems.

The Eternal system [17] uses the interception technique, in which system calls that corresponds to IIOP invocations are intercepted, and turned to a group communication system for replication. This approach is the most transparent one, as it does not require any changes to the ORB, and allows turn several copies of a non replicated object into a replicated one, and allows clients to invoke methods on replicated objects transparently. The downside of this approach is that it is operating system dependent. Also, complete transparency may sometimes be a source of inconsistency in the system, if certain operations are not deterministic, e.g., depend on local clocks.

OpenDreams [6] takes yet another approach to combining CORBA and group communication systems, by introducing the notion of replication services. In this approach replicated objects are accessed by invoking a method on a replication service, which takes care of forwarding the request to all replicas, and then merging the replies. Smart proxies can be used with this approach to hide the replication service from clients that are oblivious to replication. The downside of this approach is the extra overhead in accessing replicated objects. In this project we explore a somewhat similar approach to OpenDreams, although in our approach we eliminate the need for replication services altogether by using smart proxies, as described in [6].

User-level software-only distributed shared memory (DSM) systems, such as Millipede [12, 13], are another example of systems that need to migrate code in a cluster of workstations environment.
In particular, in Millipede both code and data can migrate between machines based on their load and on access patterns to shared memory pages. Symphony complements Millipede’s functionality, as Symphony provides services for replicating code, and for finding candidate machines for migrating code based on their load and communication capabilities. We are currently planning to port Millipede to use Symphony’s services. After the port is complete, Millipede will still manage its own memory coherency and intra-process communication, but will rely on Symphony for detecting candidate machines and starting Millipede daemons there, and will use Symphony’s informational services, such as load and topology, and other basic services such as naming.

Condor [5] and Utopia/LSF [20] are systems for doing batch scheduling in distributed environments. They focus on queuing many independent computation tasks and distributing them among a large network, whenever idle machines are detected. Since both these systems only handle independent computation tasks, they do not need to interact with the applications they run. Symphony could aid in the development of batch queuing systems like Condor and Utopia/LSF since it provides the basic services required by systems. For example, finding appropriate machines and running a code there is provided by Symphony’s management services.

8 Discussion

In recent years many organizations shift from stand-alone personal computers that, perhaps, share a printer or a file server, to a networked environment, in which computers can share and exchange information, and can cooperate in computing resource intensive tasks. Moreover, with the advent of the Internet, most computers worldwide are nowadays interconnected. While the main motivation behind this trend is to be able to share and access remote information, we propose a way of generating added value from being interconnected.

Our system allows organizations to share their computing resources at a global level, in order to maximize their utilization. By using Symphony, an organization can reduce its computing budget and improve its customer service at the same time; users of Symphony need only purchase enough computing power to handle their normal computing needs, and rely on being able to use remote computers when faced with a sudden hike in demand for their services.

On the other hand, Symphony is not a distributed operating system, in the sense that it does not attempt to replace local operating systems. Instead, Symphony complements the local operating system by adding remote execution capabilities and global load balancing. In particular, Symphony does not interfere with local scheduling policies, files systems management, etc., and has no jurisdiction over local processes.

Finally, the CORBA style service-based architecture of Symphony is important in achieving our goals of scalability, interoperability, and extensibility. Also, the combination of CORBA and group communication technology is extremely instrumental in implementing a reliable scalable system.

The holy grail of distributed computing is achieving a programming model that is general enough for writing any kind of distributed application, hides distribution and replication from programmers, yet is efficient and maintains consistency among object replicas. Symphony falls short of achieving this goal; while we offer CORBA interfaces and group communication, in these programming paradigms replication is not always transparent, application developers may choose to keep replicas inconsistent, and especially with CORBA, the performance is not always as good as one could hope for. Another part of this project deals with devising advanced programming models; the results of which are reported in separate papers. Also, as discussed in Section 7, Symphony was developed with the notion of virtual servers in mind, and is designed to handle issues related to virtual servers. Other types of distributed applications, e.g., mobile agents, may have different
needs that are not properly addressed by Symphony.

Acknowledgements: We wish to thank all other participants in this project, including Tal Anker, Yossi Ben-Asher, Danny Dolev, Dror Feitelson, Michael Rodeh, Ohad Rodeh, and Roman Vitenberg.
References


