Chapter 1

UTILITY COMPUTING FOR INTERNET APPLICATIONS

Claudia Canali
University of Parma
Parco Area delle Scienze 181A, 43100 Parma, Italy
claudia@samba.ing.unimo.it

Michael Rabinovich
AT&T Labs - Research
180 Park Avenue, Florham Park, NJ 07932
misha@research.att.com

Zhen Xiao
AT&T Labs - Research
180 Park Avenue, Florham Park, NJ 07932
xiao@research.att.com

Abstract With the growing demand for computing resources and network capacity, providing scalable and reliable computing service on the Internet becomes a challenging problem. Recently, much attention has been paid to the "utility computing" concept that aims to provide computing as a utility service similar to water and electricity. While the concept is very challenging in general, we focus our attention in this chapter to a restrictive environment - Web applications. Given the ubiquitous use of Web applications on the Internet, this environment is rich and important enough to warrant careful research. This chapter describes the approaches and challenges related to the architecture and algorithm design in building such a computing platform.

Keywords: utility computing, Web applications, application servers, resource provisioning

1. Introduction

The past decade saw an exponential growth in the amount of computing power, storage capacity, and network bandwidth. Various applications have been developed that utilize the available computing resources to better serve our society. For business big or small, the IT spending usually constitutes a substantial part of the overall budget. For ordinary people, our daily lives have become more and more computerized: we use computers to communicate with friends, read news, manage finance, etc.

As the demand for computing is increasingly becoming an integral part of our daily lives, a challenge that arises is how to provide reliable, highly available, and affordable computing service to everyone. The computing model as it exists today is far from satisfying these needs. With the rapid decrease of hardware expense, the labor costs have become a major expense in IT spending: the typical monthly salary of a system administrator is higher than the price of a group of high-end PCs. Moreover, as computer systems grow more and more complex, the expertise required to maintain such systems increases rapidly. It can be expensive for many businesses to retain the necessary computer staff. For ordinary home users, dedicated system support is seldom available. If a machine crashes, hangs, or is infected by viruses, significant effort is needed to remedy the situation.

Ideally, we want computing service to be provided like a utility service similar to water and electricity. Clearly, we understand the convenience of existing utility services: if we want to get water, we only need to turn on the tap. Imagine that the electric wires or water pipes in your home had to be “patched” or “rebooted” every so often! A similar approach to computing, where maintenance complexity is outsourced and maintenance costs are shared across a number of customers, would be very desirable.

While realizing the utility computing concept is a very challenging problem in general, we restrict our scope in this chapter to Web applications. Given the ubiquitous use of Web applications on the Internet, we believe that focusing on these applications is an important step towards providing a more generic utility computing service.

We can envision the Web-based utility computing model as a specialized distributed system with “transparency” being a major design objective. Under this model, a business can adjust the resource provisioning for its Internet applications dynamically based on user demand. For example, if an application suddenly becomes popular and receives a large number of user requests, the application service provider will allocate more servers for this application automatically to accommodate the surge in demand. Later, when the popularity of the application decreases, these servers can be re-purposed for other applications. Such a computing model can achieve a much higher degree of ef-

ficiency in resource utilization than a traditional model where each application has its dedicated infrastructure. Moreover, the locations of the computing resources are transparent to the end users. In other words, a service provider can optimize its resource provisioning based on any criteria it considers important (e.g. load, proximity, pricing, fault-tolerance, etc.) as long as the end users are provided with a single image of reliable, uninterrupted service.

There are several issues that need to be addressed in developing a utility computing system for Web applications:

- **Application distribution**: How does the system distribute instances of the application across its network? Also, how does the system maintain the consistency of different replicas of an application?

- **Resource isolation**: Given that the platform runs multiple applications on a shared platform, how can the applications be isolated from affecting each other? For example, if one application becomes a victim to a denial of service attack, this should not affect other applications.

- **Application placement**: How much resources should be allocated to an application and where (i.e., in which nodes within the platform) should the application instances be placed? A variety of metrics can be considered here, including: user proximity, server load, billing, security, relocation cost, customer preferences, DNS caching effect[1], etc..

- **Request distribution**: When a user sends a request, where should the system direct the request? A naive approach is to always distribute requests to the closest application servers. However, as we will see later, consideration of other factors is necessary for the stability of the system.

The rest of the chapter is organized as follows. We first give an overview of related work in Section 2. Then we describe a typical architecture for Web applications in Section 3 and explore some alternative architectures for utility computing in Section 4. Section 5 details our framework that allows sharing of servers at the application level. Server clusters and the maintenance of session state are discussed in Section 6 and 7, respectively. Section 8 discusses algorithmic challenges and, in particular, the danger of vicious cycles. Section 9 concludes.

2. Related Work

Related research has been conducted in Web caching, content delivery, and other distributed system areas. The literature is vast on caching and replication

[1] As we will see later in the chapter, residual requests due to DNS caching may hinder the removal of the last replica for an application in a data center.

of Web objects, consistency maintenance of different replicas, and efficient generation of dynamic responses, with too many sources to cite here. Of special interest is work on “Edge Computing”, where the computation is pushed to the edge of the network for better efficiency and performance. For example, several proposals assemble a Web response dynamically at the edge from static and dynamic components [7, 8]. Further, the authors and others have developed the *Client-Side Includes* mechanism that pushes the dynamic assembly of the Web response all the way to the browser [21]. As another example, Globule is an object-oriented system that encapsulates content into special Globule objects for replication [18]. Each object can specify its own policy on distribution and consistency maintenance.

Research in grid computing focuses on how to utilize networked computing resources in an efficient manner [10]. It shares many common objectives with utility computing. In fact, an approach for implementing scalable Web services using grid technology is described in [23]. Both grid and utility computing represent important directions for further research. Grid technology stresses support for sharing computing resources contributed by different organizations. Utility computing has a special emphasis on transparency and convenience, characteristics necessary for common utility services. An important goal in utility computing is to offload the management of computing resources from individual businesses to a separate organization. Similar to a public utility company that generates electric power and delivers it though the wires, a utility computing company manages the computing resources (e.g. CPU, memory, storage, etc.) in a number of nodes (often referred to as “data centers” in industry) and delivers the computing power through a high speed network. Because implementing the general utility computing concept is difficult, existing approaches concentrate on providing specialized utility services for particular resources, such as storage or database.

In our work, we are exploring a utility computing service that targets a particular class of applications, namely, the Internet applications. The specifics of these applications simplify the problem, yet they are so widely used that a solution targeting just these applications is worthwhile and interesting in its own right. The following are the characteristics of Internet applications that we exploit:

- Client-server computing. Internet applications are inherently client-server. Invocations of an application never result in long-running tasks. These invocation define natural boundaries in computation, and we can limit resource reassignment to these boundaries. Thus, when reassigning resources between applications, we do not need to migrate running tasks, but rather install new instances of an application and uninstall old instances of applications.

- Single carrier. Unlike general grid, we assume that the entire utility computing platform is operated by a single organization which is trusted by the application provider. This simplifies the issues of security and privacy that are challenging in general grid computing [10]. This also simplifies the accounting of resource usage and billing.

- Tiered architecture. Internet applications typically use a tiered architecture that relies on centralized core database servers (see Section 3). This simplifies data consistency issues.

3. A Tiered Architecture of Web Applications

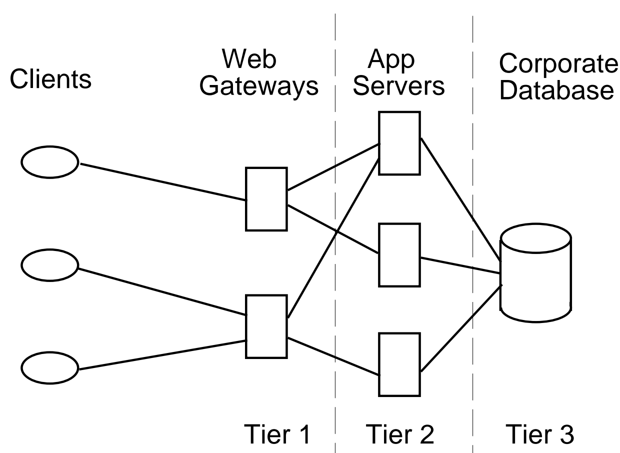

Figure 1.1. A three tier architecture for Internet applications

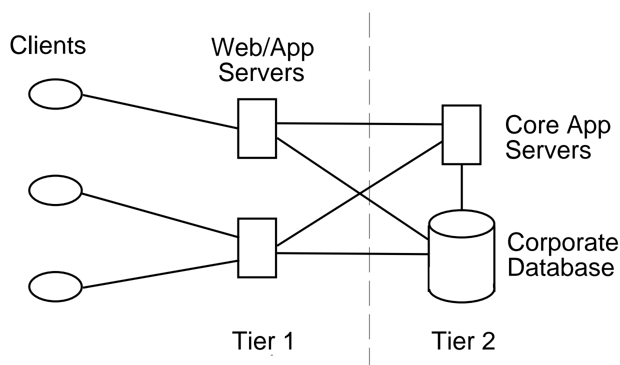

Figure 1.2. A two-tier architecture for Internet applications

Web applications are usually implemented in a tiered fashion. Typically, three tiers are distinguished as illustrated in Figure 1.1: the first tier includes Web gateways that accepts HTTP requests from users, the second tier contains application servers which execute the application code, and the third tier is database servers containing company data accessed or recorded by the application. Often, the first tier servers act as both Web gateways and applications servers and thus assume some functionality of the second tier. Therefore, the system can be often viewed as consisting of just two tiers: the utility computing tier and the back-end servers tier. This architecture is depicted on Figure 1.2. The utility computing tier includes servers that are operated by the utility computing platform, with the common resources shared appropriately between different services. The utility computing tier includes first and some of the second tier servers. The back-end tier includes servers that are outside the scope of the utility computing platform, with statically allocated resources. These typically include back-end database servers (which are often located on customer sites) and can also include some application servers.

Note that back-end database servers themselves could use some sort of dynamic resource sharing. However, this sharing is done using different mechanisms and is outside the scope of this paper. An approach to improve scalability of back-end databases is described in [22].

4. Architectural Alternatives for Utility Computing

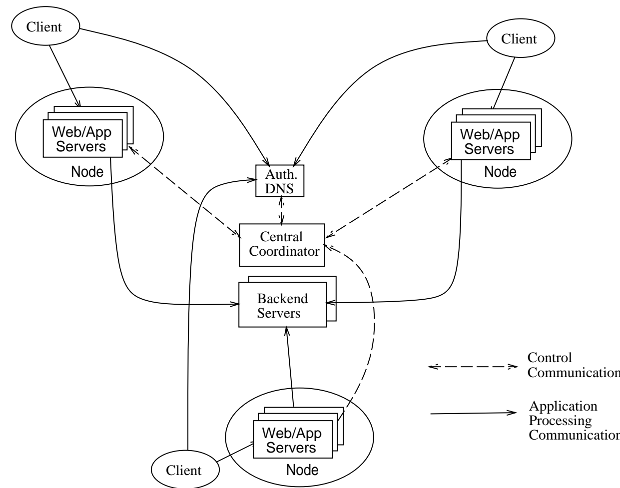

Figure 1.3. A utility computing platform for Internet applications: high-level view

The general architecture at a high level is similar to a traditional content delivery network (CDN) and is shown in Figure 1.3. It includes a request

distribution component which routes client requests to a particular application server, the utility computing tier of servers, and back-end servers. We assume that DNS-based request distribution, the prevalent mechanism in traditional CDNs, is used here as well. (There are multiple mechanisms for request distribution. See [4] for a survey.) Consequently, a Web application is identified by the host name of its URLs, with specific requests and parameters forming the remaining parts of the URLs. For example, we may have an on-line stock broker application www.SellStocks.com and a specific request https://www.SellStocks.com/buy?ticker=T.

Before invoking the application, a client needs to resolve the host name, www.SellStocks.com in our example, to an Internet address. Thus, the client sends a DNS query to the platform's authoritative DNS server, which chooses an appropriate server for this query and responds with the corresponding IP address. The client will then contact the chosen application server. Note that due to DNS response caching, several subsequent HTTP requests to this service from the same client, as well as from other clients using the same client DNS server, will also go to the same server. The server processes the request, contacting backend servers using secure communication such as IPsec [13]. In parallel, each server monitors the demand and performance of its applications. The particular architecture shown contains the central coordinator that collects these measurements and makes periodic policy decisions on replica placement and request distribution. According to these decisions, the coordinator sends instructions to application servers to install or uninstall applications, and request distribution rules to the authoritative DNS server. The coordinator may have other functions such as maintaining consistency of application replicas by propagating updates from the primary server. The specific functionality of the coordinator can vary because some functionality can be implemented in a distributed fashion by application servers directly [20].

Within this general framework, there are several alternatives for architecting the system, depending on the granularity of resource sharing and on the mechanism for request routing between application replicas within one platform node.

In terms of requests routing mechanisms within a node, one possibility is to allow the DNS server select individual application servers when responding to client DNS queries, and send client requests to these servers directly. This is the alternative shown in Figure 1.3. Another possibility is to connect the application servers at each node to a *load-balancing switch* [19]. The authoritative DNS server in this case selects only a node and sends clients to the corresponding switch. The switch then load-balances received requests among servers that have the requested application.

Turning to resource sharing, one can achieve fine-grained resource sharing when the same application server (such as Tomcat [2]) is shared by multiple

applications. In this case, applications are installed by simply placing them into appropriate directories on the application server. On the other extreme, resource sharing may occur at the granularity of the whole servers, so each server may belong to only one application at a time. Many application servers offer clustering capabilities which simplify implementation of this approach. Finally, one can use server-grained allocation and clustering technologies as above, but employ virtual machine monitors to run multiple virtual application servers on the same physical machine. We consider these alternatives in more detail next.

5. Application Server Sharing

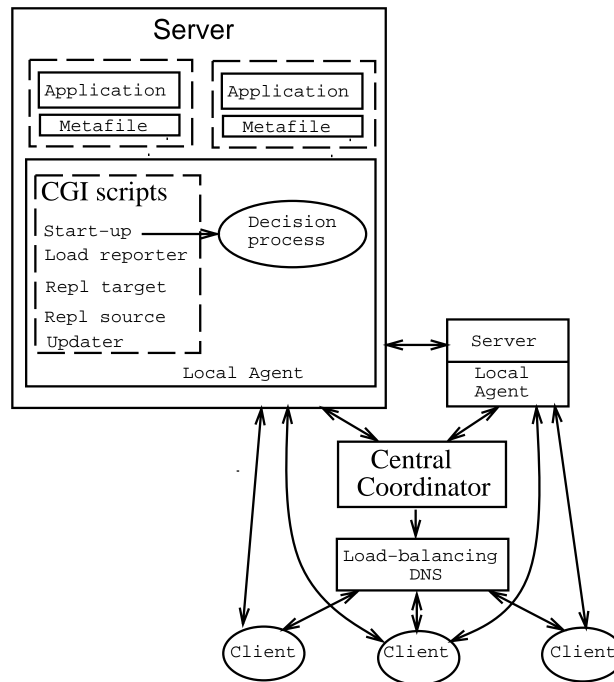

Figure 1.4. ACDN architecture

An application server can be configured to host multiple applications. In this approach, creating an application replica requires simply putting the application in an appropriate directory on the application server. We implemented a prototype of this approach in our ACDN system [16, 20]. In particular, our implementation showed that the entire system can be implemented on top of the standard HTTP protocol and off-the shelf application servers. (We used Apache [1] in our prototype, with Fast CGI [9] as the implementation mecha-

nism for applications.) Not only does this simplify the adoption of the system, but it also allows easy traversal of firewalls and network address translation devices, and hence permits the deployment of the system over public Internet.

The architecture of the system is shown in Figure 1.4. The system operates in cooperation between the central coordinator and local agents deployed on individual servers. The local agents implement functionalities related to application distribution. They contain sets of CGI scripts, and so are a pure add-on to any standard Web server. Before starting to use a server, the central coordinator invokes its *start-up script*. This script forks a *decision process* that periodically examines every application on the server and decides if any of them must be replicated or deleted. This is due to the fact that replica placement decisions in ACDN are distributed among individual servers: each server makes decisions about the applications that it currently deploys based on its own local usage.

The global central coordinator keeps track of application replicas in the system, collects periodic load reports from servers (by periodically invoking the *load reporter script* on each server), recomputes request distribution policy, communicates this policy to the load-balancing DNS server, and maintains application replicas on all servers consistent with the primary copy (by periodically invoking the *updater script* on each server). The coordinator also gives permission to servers to delete underused application replicas, which ensures that at least one replica of each application remains in the system even in the face of simultaneous decisions to delete the application by all servers. Finally, the central coordinator also answers queries from servers for the least-loaded server in the system. The servers use this information in some of their replica placement decisions.

Although the central coordinator is theoretically a bottleneck, this is not a concern in practice since the amount of processing it does is minimal; it can in fact be physically co-located with the DNS server. The central coordinator is also a single point of failure. However, it is not involved in processing user requests. Thus, its failure only leads to a stop in application migrations or replications and does not affect the processing of requests by the existing replicas. Furthermore, the central coordinator's state can be reconstructed upon its recovery without a halt to processing of user requests.

5.1 Application Distribution Framework

We are not aware of any off-the-shelf support for the application distribution framework with application server sharing. Thus, the system must implement the entire framework itself. The ACDN implementation is based on the concept of a metafile, inspired by ideas from the area of software distribution such as the technology by Marimba Corp. [11].

```
<FILE>
  /home/apps/maps/query-engine.cgi 1999.apr.14.08:46:12
</FILE>
<FILE>
  /home/apps/maps/map-database 2000.oct.15.13:15:59
</FILE>
<FILE>
  /home/apps/maps/user-preferences 2001.jan.30.18:00:05
</FILE>
<SCRIPT>
  mkdir /home/apps/mapping/access-stats
  setenv ACCESS_DIRECTORY /home/apps/maps/access-stats
</SCRIPT>
```

Figure 1.5. An example of a metafile

Conceptually, the metafile consists of two parts: the list of all files comprising the application along with their last-modified dates; and the *initialization script* that the recipient server must run before accepting any requests. Figure 1.5 provides an example of a metafile that represents a map-drawing application consisting of three files: an executable file that is invoked on access to this application and two data files used by this executable to generate responses. The metafile also contains the initialization script that creates a directory where the application collects usage statistics and sets the corresponding environment variable used by the executable. The initialization script can be an arbitrary shell script. When the initialization script is large, the metafile can include just a URL of the file containing the script. The metafile is treated as any other static Web page and has its own URL.

Using the metafile, all the tasks of the application distribution framework can be implemented over standard HTTP. The process of replica creation is initiated by the decision process on an ACDN server with an existing replica. It involves the *replication target script* on the target server and the *replication source script* at the source server. Once the replication is complete, the replica target script informs the central coordinator about the new application replica. The central coordinator recomputes its request distribution policy based on the new replica set and sends the new policy to the DNS server.

Replica deletion is initiated by the decision process on a server with the application replica. This server first must notify the central coordinator of its intention to delete its replica. The coordinator in turn needs to notify the DNS server to exclude this replica from its request distribution policy. After that, the coordinator can grant the ACDN server the permission to delete the replica. The actual deletion, however, happens after a delay corresponding to the DNS time-to-live (TTL) value associated with the domain name of the application.

This is to accommodate residual requests that might arrive due to earlier DNS responses still cached by the clients.

Application migration can be implemented as replication followed by deletion. We will describe consistency maintenance in the following subsection. More details on these tasks in application distribution can be found in [20].

5.2 Consistency Maintenance

Maintaining consistency of application replicas is important for the correct functioning of our system. There are three problems related to replica consistency:

- **Replica coherency:** An application typically contains multiple components whose versions must be mutually consistent for the application to function properly. A replica for an application can become incoherent if it acquired updates to some of its files but not others so that there is a version mismatch among individual files.
- **Replica staleness:** A replica for an application can become stale if it missed some updates to the application.
- **Replica divergence:** Replica divergence occurs when multiple replicas for an application receive conflicting updates at the same time.

The metafile described earlier provides an effective solution to the replica staleness and coherency problems. With the metafile, whenever some objects in the application change, the application's primary server updates the metafile accordingly. Other servers can update their copies of the application by downloading the new metafile as well as all modified objects described in the metafile. This ensures that they always get the coherent new version of the application. Thus, the metafile reduces the application staleness and coherency problems to the cache consistency of an individual static page – the metafile.

In our system, application updates are performed asynchronously with requests arrivals: a server keeps using the current version of an application to serve user requests until the new version is ready. This is to avoid a prohibitive delay to the user perceived latency that will otherwise occur. If an application has stringent consistency requirements, it may use application level redirection to redirect user requests to the server with the most recent version of the application.

The final problem related to replica consistency is replica divergence. It happens when an application is updated simultaneously at multiple replicas. In general, there are two types of updates:

- **Developer updates:** These are updates introduced by the application authors or maintainers. It can involve changes to the application code (e.g.

software upgrades or patches) or to the underlying data (e.g. updates of product pricing).

- **User updates:** These are updates that occur as a result of user accesses (e.g. updates during e-commerce transactions). It involves changes to the data only.

Our system avoids replica divergence by restricting developer updates to only one application replica at any given time. This replica is called the *primary* replica for the application. It ensures that all the updates to the application can be properly serialized. The coordinator keeps track of the location of the primary replica for each application. The idea is similar to classic token-based schemes where a token circulates among a group of servers and only the server that holds the token can perform updates. For updates that occur as a result of user accesses, we either assume they can be merged periodically off-line (which is the case for commutative updates such as access logs) or that these updates are done on a shared back-end database and hence they do not violate replica consistency.

5.3 Discussion

One advantage of application server sharing is that replication of an application is often very quick. Indeed, unless the deployment of a new application instance requires server restart (which may happen when the application requires server reconfiguration, such as changing a log file), it involves little more than just copying and unpacking a tar file with the application components. In our prototype, this takes in the order of single seconds. Another advantage is the efficient utilization of server resources. When one application does not utilize the server, spare capacity can be used by another application.

Unfortunately, application server sharing also has a very serious drawback, namely, poor resource isolation. Different applications sharing the same server can affect each other in a variety of undesirable ways. If one application falls a victim to a denial of service attack, other applications on the same server will be affected. If one application fails and manages to hang the computer, other applications will be affected (this is referred to as fault isolation). No matter how well an application is hardened against hacker break-ins, its resiliency may be affected by a less secure application that shares the same server. Finally, billing and resource usage accounting is more complicated with server sharing. Although much work has been done on addressing these issues, they continue to impede this approach.

6. Server Clusters

To avoid the poor resource isolation of application server sharing, the platform can always run each application instance on its own dedicated application server. The dedicated server approach further allows two main alternatives, depending on whether or not each application server runs on a dedicated physical machine. We consider these alternatives next.

6.1 Physical Server Clusters

In this approach, the system runs each application replica on a dedicated application server, and each application server on a dedicated physical machine. In other words, the system performs resource allocation at physical server granularity. This scheme allows almost absolute resource isolation. Furthermore, this approach is well-supported by existing server cluster technologies including WebLogic [3], WebSphere [12], and Oracle's Application Server 10g [17]. In particular, server clusters implement consistency management for the application, and quick fail-over capabilities with session state replication discussed later in Section 7. For example, WebLogic allows all replicas of an application that run within the same node to form a cluster, and all clusters of this application to form a single *domain*. Then WebLogic can ensure the consistency of all the application replicas in the domain, while performing fast fail-overs only among servers within each cluster.

On the other hand, this scheme is characterized by slow resource allocation. Indeed, moving a physical machine from one application cluster to another involves restarting the application server on this machine in the new cluster, which may take around a minute in our experience on a popular commercial application server. Another disadvantage of this scheme is that it may result in poor resource utilization. An application replica ties up an entire physical machine. So if this application does not fully utilizes the machine, the spare capacity is wasted.

6.2 Virtual Server Clusters

One can try to combine the advantages of the application server sharing and physical server sharing by utilizing the virtual machine technology. This technology allows several *virtual machines* to share the same physical machine. Each virtual machine provides an illusion of a dedicated machine to the operating system and any application running on it. In fact, different virtual machines sharing the same physical machine can run different operating systems. The operating systems running on the virtual machines are called *guest* operating systems, and the operating system that runs directly on the physical machine and enables the virtual machine sharing is called the *host* operating system.

With the virtual machine technology, one can still run each application replica on a dedicated application server, and use existing clustering technologies to facilitate replication and fail-over. However, each application server can now run on a virtual rather than a physical machine. Virtual machines allow very good resource and fault isolation, and at the same time permit good resource utilization, because the physical resources are shared among virtual machines (and hence applications).

Yet this approach is not without its own limitations. A technical issue is the overhead of running a guest operating system on top of the host operating system. A study using an open source implementation of a virtual machine monitor found that system calls on a Linux guest OS running on top of a Linux host OS experienced up to a factor of 30 slowdown [15]. The same study found that sample applications ran 30-50% slower in this environment. Commercial virtual machine implementations such as VMWare apparently are more efficient, with performance penalties within single percentage points on standard benchmarks [6]. However, these products are expensive – the price of a VMWare license can be higher than that of a typical physical machine. Thus, the economy from better resource utilization can only be realized if a significant number of virtual machines can be multiplexed on the same physical server.

7. Session State Maintenance

Many Internet applications require prolonged sessions between the client and the server, which span multiple HTTP interactions. Figure 1.6 shows an example of a user browsing an on-line travel site. A session in this case may involve a sequence of interactions including flight and hotel inquiry, filling out the itinerary and customer profile, and finally completing the payment. Each inquiry depends on the context of the session, and the server must maintain session state from one interaction to the next. In the simplest case, this state can be maintained on the client using cookies. However, in most cases, the session state is too large or too valuable to hand off to the client, and servers maintain it themselves.

The need to maintain session state poses two challenges to the utility computing platform. First, for the duration of the session, requests from the same client must go to the same server that holds the session state. We refer to this behavior as *client stickiness*. Second, fulfilling the promise of high reliability requires the recovery of the session state from a failed server. Being able to move from a failed to an operational server in the middle of a session is referred to as *session fail-over*.

The basic idea to address both problems is the same: it involves encoding the identity of the server that has the session state into the response to the

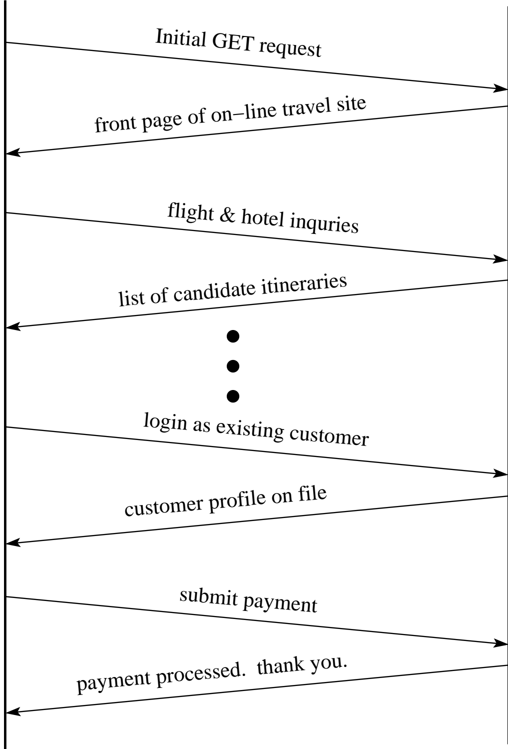

Figure 1.6. An example session of an on-line travel Web site.

client and making the client include this information in subsequent requests. The easiest way to accomplish this is through the use of cookies as illustrated in Figure 1.7 (top diagram): when a server receives the first request of the session, it chooses a secondary server with the application and replicates the session state to the secondary server. In response to the client, the original (also called the primary) server sets a cookie, which records the identity of both the primary and secondary servers for this session.

Because clients cache DNS responses and load-balancing switches (if used) try to select the same server for the same client, client stickiness will be upheld most of the time. However, should a different server receive a subsequent request in the session, the request's cookie will have all the information to resolve the situation. If the original server is up, the new server can use an HTTP redirect to send the client to the original server and thus enforce client stick-

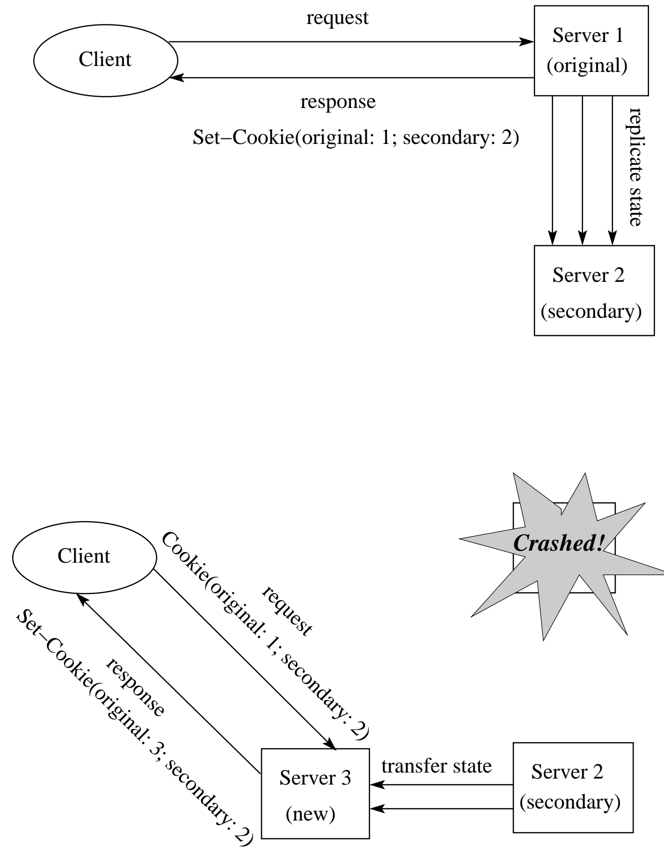

Figure 1.7. Support for client stickiness and session fail-over.

iness.[2] If the original server is down, the new server can implement session fail-over by obtaining the session state from the secondary server and taking over processing of this and all subsequent requests for this session. This is illustrated in the bottom diagram of Figure 1.7. (The cookie is modified accordingly to record the identity of the new server in place of the original server.) Because session fail-over requires tight communication between the original and the secondary servers for continuous state updates, it is typically only used between servers in the same site.

[2]The only subtlety is that the new server must specify the (virtual) IP address of the original server in the redirect response. Indeed, if the new server used the host name for the redirection, the client would have to resolve this name again, and the cycle would repeat.

8. Algorithmic Challenges

As already mentioned, a utility computing platform for Internet applications involves two main algorithms: the algorithm for application replica placement and the algorithm for distributing requests among those replicas. Designing these algorithms involves several alternatives. The first choice has to do with the extent of central coordination in the algorithm. The architecture of Figure 1.3 assumes a central point for request distribution (DNS) and so it is natural to assume that the request distribution algorithm be centralized. However, the choice is less clear for the replica placement algorithm. While a centralized algorithm has a view of the entire system and can result in highly optimized placement, a distributed algorithm can be more scalable.

Within the centralized approach to replica placement, another fundamental choice is between a greedy incremental approach and a global optimization approach. The greedy approach utilizes feedback from monitoring the application to make incremental adjustments in replica placement. The global optimization approach uses monitoring results to periodically solve a mathematical optimization formulation from scratch, independent of the current configuration [14]. In our work, we explored a distributed approach to replica placement, where each server examines its own access log and decides whether any of its applications should be migrated or replicated to another server [20].

In this chapter, rather than describing a specific algorithm, we concentrate on an interesting issue faced by any such algorithm, namely, the possibility of vicious cycles. Because the issues we discuss are independent of the number of servers in each node of the platform, we assume for simplicity that each node contains only one server.

8.1 Vicious Cycles in Request Distribution

As an example of vicious cycles in request distribution, consider a natural request distribution algorithm as follows:

> The authoritative DNS server resolves a query to the closest non-overloaded server, where overload is determined by a load threshold, e.g., 80% server utilization.

Assume that the system periodically reconsiders the request distribution policy based on performance observed prior to the decision. This simple and intuitive algorithm can easily lead to oscillations due to a herd effect [5]. Consider, for example, two servers with an application replica and assume that all demand comes from the vicinity of server 1 (see Figure 1.8, top diagram). At decision time $t1$, the system realizes that server 1's utilization is over 80% and stops using it for future requests. Consequently, utilization of server 1 declines and by the next decision time $t2$ drops below 80%. At time $t2$ the system finds that server 1 is no longer overloaded and starts using it again for all future re-

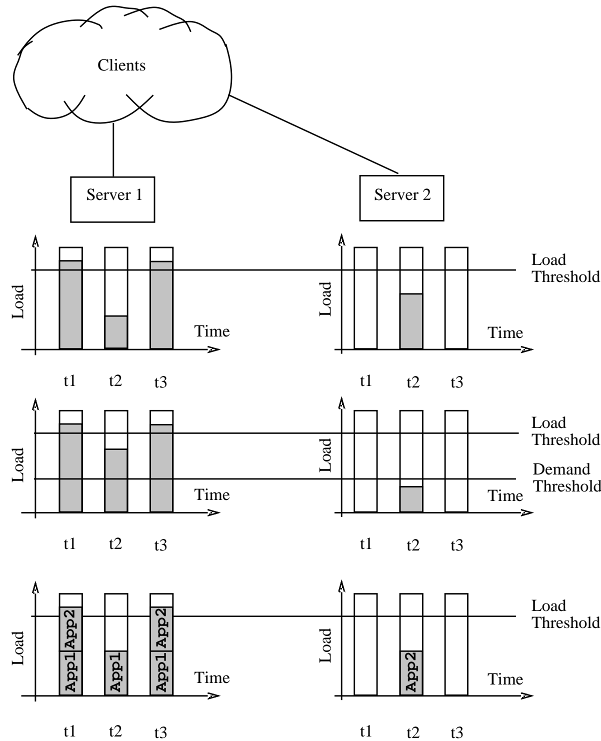

Figure 1.8. Vicious cycle examples in request distribution (top diagram), replication (middle diagram), and migration (bottom diagram).

quests. Its load increases and by the next decision time exceeds 80%, and the cycle repeats. As a result, the request distribution is never optimal: either too many requests are sent to an overloaded server 1, or too many requests are sent to a distant server 2.

8.2 Vicious Cycles in Replica Placement

Replica placement algorithms also face a danger of vicious cycles, even with resource allocation at the whole server granularity. Assume for example that content providers are billed based on resource usage and, in particular, on server usage, a natural billing model for the utility computing paradigm. Then, customers will want to keep the number of application replicas at the minimum, and it is natural for them to specify how much demand for a given application replica warrants the extra costs of having the replica. We will refer to this demand as the *demand threshold*. At the same time, customers do not want the servers running their application to be overloaded, and hence it is

natural for them to specify (or rely on the platform's default values of) a load of existing servers that should trigger the deployment of additional replicas. We will refer to this load as the *load threshold*. The demand threshold is expressed in such units as request rate or traffic volume, and the load threshold in CPU or memory utilization. Normally, the load threshold governs overload-induced replication and demand threshold affects the proximity-induced replication.

However, if the two thresholds are not carefully coordinated, the scenario illustrated on Figure 1.8 (middle diagram) may occur. As in the previous example, assume that all demand comes from the proximity of server 1, and the demand overloads this server. Then, the algorithm may trigger replication. If no spare server is available in server 1's platform node, the new replica may be created elsewhere. However, because the new replica is more distant to the demand, the request distribution favors the old replica and sends only excessive load to server 2. Then, depending on the computational requirements of the application, the excessive load may translate to the request rate that is below the demand threshold. If this happens, the replica placement algorithm can later drop the new replica, reverting to the initial configuration.

The bottom diagram in Figure 1.8 illustrates another vicious cycle that may occur in a platform with shared servers. Again, consider a platform with two servers, server 1 and 2, where server 1 is shared between two applications. Assume all demand for both applications comes from server 1's own region, and its overall load is above the load threshold. Further, consider the case where the demand on each application separately does not exceed the demand threshold, and so replication is not possible. To offload the server, the system may resort to migration, for example, moving application 1 to server 2. Once this is done, neither server is overloaded. Then, the next time the application placement is considered, the system may migrate application 1 back to server 1 because this would improve client proximity.

To prevent such cycles, the replica placement algorithm must predict how dropping a replica would affect the load on remaining servers in the second example above, or how migrating a replica would affect the load on the recipient server in the last example. Because the load on individual servers depends on request distribution algorithm, direct load prediction introduces an interdependency between the request distribution and replica placement algorithms. Such prediction is especially difficult for fine-grained server sharing. Indeed, the load imposed by an application replica on its server depends on the load distribution among all application replicas, and the load distribution depends on the aggregate server load. Thus, the prediction of load due to one application may depend on other applications sharing the same servers.

9. Conclusion

This chapter focuses on the issues and challenges in building a utility computing service for Web applications. We described a typical architecture for Web applications and discussed some alternatives in designing a utility computing platform for running these applications. In particular, we discussed tradeoffs in various approaches to resources sharing in this environment, and the issue of session state maintenance, the issue that is very important in practice but often overlooked in research in the Web performance arena. We also presented some fundamental algorithm issues in building such a platform, namely, how to avoid vicious cycles during request distribution and application replication.

References

[1] The Apache HTTP Server Project. `http://httpd.apache.org`.

[2] The Apache Jakarta Project. `http://jakarta.apache.org/tomcat`.

[3] BEA Web Logic. `http://www.bea.com`.

[4] Valeria Cardellini, Michele Colajanni, and Philip S. Yu. Dynamic load balancing on web-server systems. *IEEE Internet Computing*, 3(3):28–39, 1999.

[5] M. Dahlin. Interpreting stale load information. *IEEE Transactions on Parallel and Distributed Systems*, 11(10):1033–1047, October 2000.

[6] Murthy Devaraconda. Personal Communication.

[7] Fred Douglis, Antonio Haro, and Michael Rabinovich. HPP: HTML macro-preprocessing to support dynamic document caching. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, pages 83–94, December 1997.

[8] ESI–Accelerating E-business Applications. `http://www.esi.org/`.

[9] FastCGI. `http://www.fastcgi.com/`.

[10] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *International J. Supercomputer Applications*, 15(3), 2001.

[11] A. Van Hoff, J. Payne, and S. Shaio. Method for the distribution of code and data updates. U.S. Patent Number 5,919,247, July 6 1999.

[12] IBM WebSphere Software Platform. `http://www-306.ibm.com/software/info1/websphere`.

[13] IPsec. `http://www.ietf.org/html.charters/ipsec-charter.html`.

[14] Kamal Jain and Vijay V. Vazirani. Approximation algorithms for metric facility location and k-median problems using the primal-dual schema and lagrangian relaxation. *Journal of the ACM*, March 2001.

[15] Xuxian Jiang and Dongyan Xu. SODA: A service-on-demand architecture for application service hosting utility platforms. In *Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing*, June 2003.

[16] Pradnya Karbhari, Michael Rabinovich, Zhen Xiao, and Fred Douglis. ACDN: a content delivery network for applications. In *Proceedings of ACM SIGMOD (project demo)*, pages 619–619, June 2002.

[17] Oracle Application Server. `http://www.oracle.com/appserver`.

[18] Guillaume Pierre and Maarten van Steen. Globule: a platform for self-replicating Web documents. In *Proceedings of the 6th International Conference on Protocols for Multimedia Systems*, pages 1–11, October 2001.

[19] M. Rabinovich and O. Spatscheck. *Web Caching and Replication*. Addison-Wesley, 2001.

[20] Michael Rabinovich, Zhen Xiao, and Amit Aggarwal. Computing on the edge: A platform for replicating Internet applications. In *Proceedings of the Eighth International Workshop on Web Content Caching and Distribution*, September 2003.

[21] Michael Rabinovich, Zhen Xiao, Fred Douglis, and Chuck Kalmanek. Moving edge-side includes to the real edge—the clients. In *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems*, March 2003.

[22] Swaminathan Sivasubramanian, Gustavo Alonso, Guillaume Pierre, and Maarten van Steen. GlobeDB: Autonomic data replication for web applications. In *Proceedings of the 14th International World Wide Web Conference*, 2005.

[23] Ryoichi Ueda, Matti Hiltunen, and Richard Schlichting. Applying grid technology to Web application systems. In *Proceedings of the IEEE Conference on Cluster Computing and Grid*, May 2005.