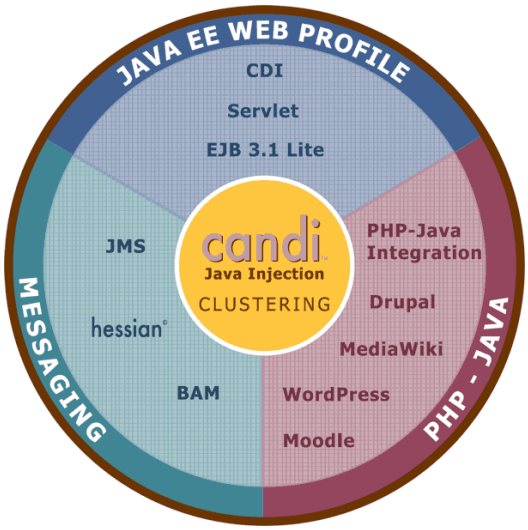
Scaling Web Applications in a Cloud Environment using Resin 4.0



Abstract

Resin 4.0 offers unprecedented support for deploying and scaling Java and PHP web applications in a cloud environment. This paper discusses the technical underpinnings of Resin 4.0's sophisticated clustering capabilities that provide reliable and fast distributed sessions, distributed object caching, and cloud-wide application deployment all while adding and removing application server instances at will during runtime.

I. Introduction

Cloud computing is an environment in which computing hardware can be dynamically reapportioned to the task at hand, usually using virtual machines. For example, one type of cloud computing environment is a cluster of physical machines maintained in-house by an organization. These physical machines all run virtual machines on which the organization's own applications will run. By using virtual machines on a number of physical machines, the organization can improve reliability and performance as well as dynamically provision the appropriate resources to various applications depending on demand.

An alternative scenario is to contract a third-party cloud computing ISP in which the organization provides virtual machine images to the ISP to run. New machine instances can be obtained from the ISP using a metered payment plan. This approach allows organizations to gain computing power on demand without the need to maintain hardware.

Both of these scenarios present a great opportunity to organizations in which the demands of its applications change either periodically or sporadically. For example, agencies that have regular deadlines for payments, applications, or registrations such as universities and government institutions may have a low level of activity on their sites during most of the year,

E sales@caucho.com T 858.456.0300
W http://caucho.com F 858.777.3636

but experience huge load in the days and weeks leading up to the deadlines. Some sites may also experience an unexpected increase load such as a news site after a breaking event. By having additional capacity available for times of high-demand, the organization can serve their clients better. At the same time, by avoiding the use of unnecessary compute power or by being able to redistribute that power to other areas, organizations can also save money and energy during low-demand periods.

Java and PHP web applications can however be difficult to maintain and deploy in such cloud environments with existing technology. To deploy an application to a set of virtualized servers, administrators need to create custom virtual machine images and distribute them to the servers. If the application code is bundled in the virtual machine image, a new image must be constructed and distributed with each new version of the application. If the application is stored on a networked storage device, it must be retrieved by each virtual machine and deployment carefully managed on updates.

The applications themselves must also adhere to certain restrictions. Without application server support, sessions will not be replicated across all machines, losing the reliability and availability advantages of replication. Object caching would require a sophisticated third-party cache framework that is fast and able to cope with the dynamic nature of a cloud environment. Thus the applications may be forced to avoid sessions and code to non-standard APIs for caching. Administrators are then also required to maintain an additional infrastructure exclusively for caching.

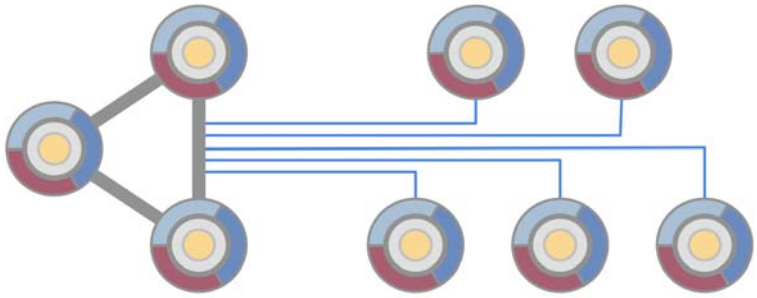
Resin 4.0 addresses all of these issues by dynamically distributing sessions, cached objects, and application files as servers are added and removed from service. Distributed sessions are transparent to both Java and PHP applications which use standard APIs. Object caching is available to applications via the standard Java Cache (JSR-107) API and the PHP APC API. Applications can be distributed to all servers in the deployment via provided plugins for tools such as ant, Maven, and Eclipse. The application will automatically be propagated to all server instances and when new instances are added, they will be brought up-to-date by the same mechanism. The loss or shutdown of server instances will not result in the loss of sessions, cached objects, or application data.

This paper describes the architecture of Resin 4.0 that makes these features possible. The optimizations that Resin 4.0 employs to make cluster-wide caching and deployment fast are also detailed. Finally, use cases and example deployment scenarios are presented to give a flavor of how Resin 4.0 scales in production environments.

E sales@caucho.com T 858.456.0300
W http://caucho.com F 858.777.3636

II. Resin 4.0 Architecture

The core part of the Resin 4.0 architecture is the **triad**, a set of three servers that provide the central repository for persistent data and maintain an up-to-date record of the dynamic servers in the system. Optimizations allow for quick access to data at any server in the system, but the triad provides a point of stability and persistence to reduce management complexity.



The triad (left) are the three core servers in a Resin 4.0 deployment. Dynamic servers (right) can be brought into the system at any time and removed at will.

The dynamic servers in the system are the workhorses for applications. Started and stopped at will, they provide elastic scaling. Each dynamic server has access to the shared data within the system via JavaEE sessions or the Java Cache API. Once properly configured, the cost of starting a new dynamic server is simply to start a new virtual machine. When a new dynamic server is brought online, it contacts one of the triad servers to announce its availability and import all application data. As applications are updated on the triad, the changes are pushed out to all the dynamic servers by the triad to keep them updated.

The dynamic servers use the triad as their persistent store for session and object cache data. At the same time, optimizations keep frequently used data in memory on the dynamic servers to improve application performance and reduce network load. Together with the triad servers, the dynamic servers form a **cluster**.

Using a combination of triad servers and dynamic servers minimizes the complexity of managing an application deployment. The triad servers are brought up first on system start up and at least one should be available at any time during the life of the system. Thus these three servers can be the main focus of administration time and effort because all of the other servers may go up or down at any time without affecting the functional performance visible to clients. Assuming a virtualized environment, if one or more triad servers become faulty at any time, a replacement or replacements can be brought into place quickly. Having three servers in a triad

E sales@caucho.com T 858.456.0300
W http://caucho.com F 858.777.3636

avoids a single point of failure, allows up to two servers to fail at any given time, and allows normal maintenance of a single server without downtime.

Resin 4.0 includes a software load balancer that distributes HTTP requests for web application clients. The triad server keeps track of the current members of the cluster and communicates with the load balancer to update it on which dynamic servers are available to handle requests. When choosing a server to handle new requests, the load balancer takes into account the CPU load of a server as well as the number of simultaneous requests that server is already handling. Depending on the algorithm selected by the administrator, the load balancer can either direct the request to the least loaded server to keep load even or to the same set of servers until they are fully loaded to avoid starting new servers. Once a server has been selected for a new request, subsequent requests from the same client will go to the same server to avoid unnecessary load times.

Large clusters

As deployment sizes grow, many organizations split their servers into different networks to improve reliability, availability, and manageability. Resin 4.0 clustering supports this partitioning explicitly by allowing a cluster to be split into **pods.**

4.0 clusters may have multiple pods to allow administrators to align their application clustering with their network partitioning for improved performance. Resin 4.0 deployment. Dynamic servers (right) can be brought into the system at any time and removed at will.

Each cluster pod contains its own triad which manages the object cache, distributed sessions, and application file repositories for a set of dynamic servers. This architecture makes it possible to create a pod

E sales@caucho.com T 858.456.0300
W http://caucho.com F 858.777.3636

for each local network within a single site or a pod for each of a number of geographically distributed sites. In either case, the dynamic servers need only to coordinate with their local triad to access cache, session, and application file data for fast retrieval and low network overhead. At the same time, all the pods are considered to be part of a single logical cluster serving the same set of applications. Thus cache data and sessions can be shared across the entire network. Moreover any applications that the administrator deploys will be propagated throughout all the pods, even at remote sites.

When configuring Resin 4.0, the administrator selects one of the pods to be a master pod. This pod maintains the authoritative copy of the data that all the other pods use for caching and applications. Thus if an application needs data not in its local pod, it is easy to find by asking the master pod's triad.

III. Distributed Caching and Sessions

Resin 4.0 introduces a new distributed caching architecture to support object caching and HTTP sessions for applications. Object caching is visible to applications via the Java Cache (JSR-107) API while HTTP sessions are part of the basic Java EE Servlet specification. From the point of view of the application developer, these facilities should be used in the same way regardless of the size of the cluster, allowing arbitrary scaling. Thus the developer is freed from ongoing infrastructure concerns, while the administrator is given the flexibility to add and remove server capacity at will.

To make caching and sessions truly transparent to the developer, a distributed caching architecture is necessary. When an application running on one server caches a value, that same value should be available to

```
package example;

import java.io.*;
import javax.servlet.*;
import javax.inject.Current;
import javax.cache.Cache;

public class MyServlet extends
GenericServlet {
  @Current Cache _cache;

  public void service(ServletRequest req,
                      ServletResponse res)
  {
     PrintWriter out = res.getWriter();

     String data =
       (String) _cache.get("my-data");

     if (data != null) {
       out.println("cached data: " + data);
     }
     else {
       data = generateComplicatedData();
       _cache.put("my-data", data);
       out.println("new data: " + data);
     }
  }
}
```

An example of using the Java Cache API. Regardless of which server this code is run on in a Resin 4.0 cluster, the behavior will be the same and the same data will be accessed. The

E sales@caucho.com T 858.456.0300
W http://caucho.com F 858.777.3636

applications running on another server so that they can all take advantage of the same cached data. However as far as the developer is concerned, he or she needs only to use method calls to get and put data without needing to know where the data is actually stored.

Distributed sessions can be handled in a similar way to improve reliability without developer intervention. Suppose that an application on one server starts a new session with a user, but that server later fails. The load balancer can then redirect the user on the next request to another server running the same application; yet retain the session data without interrupting the user's experience.

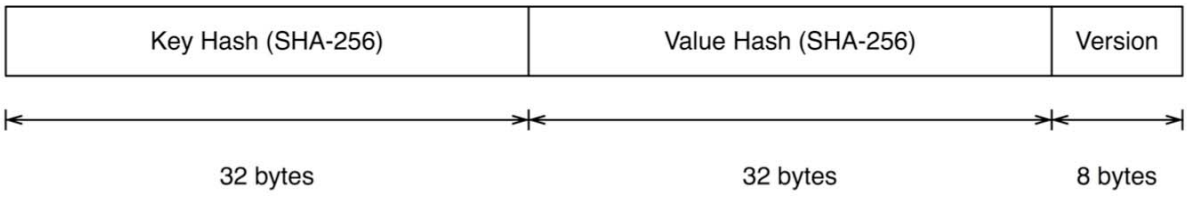
Both object caching and distributed sessions are simply types of caching with different synchronization requirements. Resin 4.0 uses a single, unified framework to handle both. The next section will describe the general framework and how each API is built using it.

Implementing Resin 4.0's distributed cache

In Resin 4.0 the triad servers handle the synchronization and storage of cached data. The

metadata used for synchronization and the actual cached data are distributed separately. This key innovation makes Resin 4.0 highly efficient in managing the cache. By decoupling the storage of the metadata from the data, Resin 4.0 is able to avoid unnecessary network communication because only the metadata needs to be compared when checking for updates to the cache. The most common pattern of using a cache in web applications is to store data that is read more often than it is written. Thus real updates to cached data are often relatively infrequent and so by sending only metadata during synchronization, the expense of sending redundant cached data is avoided.

A cache can be viewed as a map from a key to a value. Resin 4.0 uses a fixed size metadata structure called an **m-node** that includes hashes of both the cache key and the cache data as well as a version number used for synchronization purposes. To improve synchronization



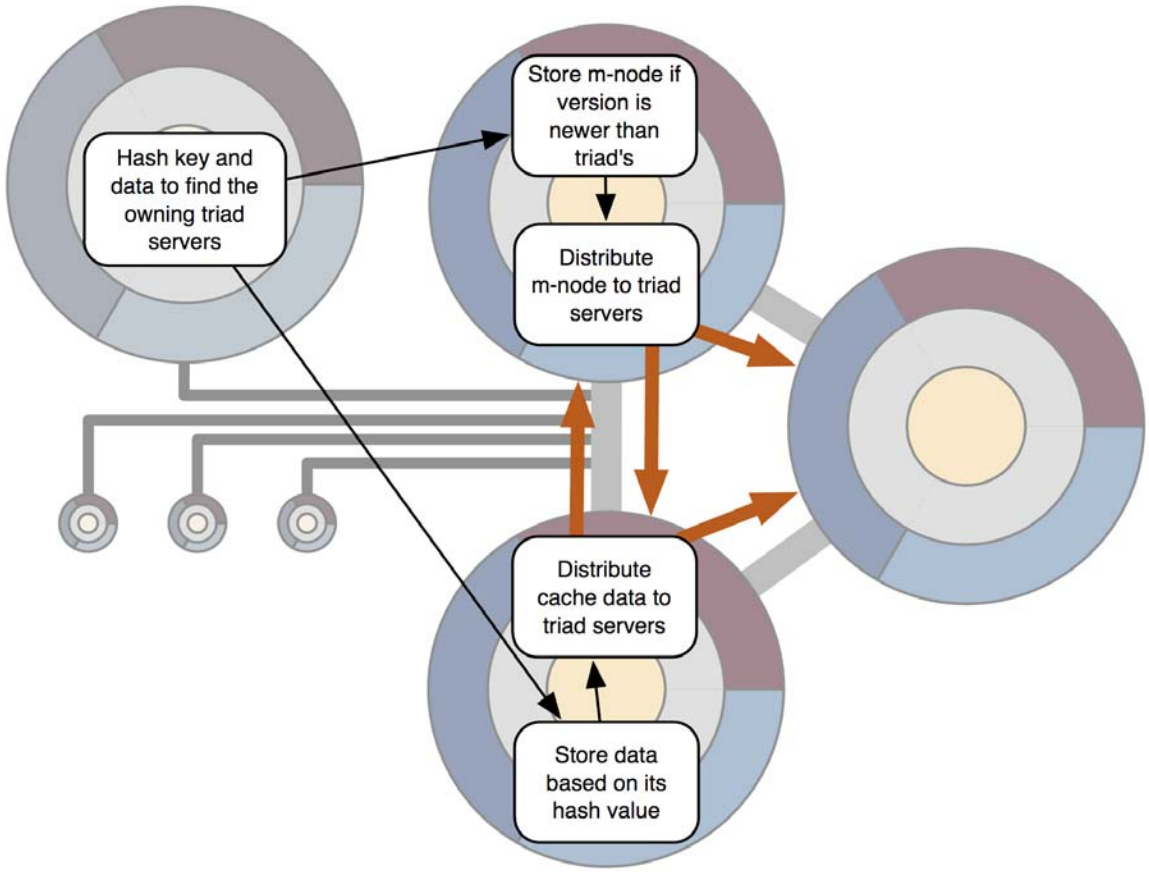
Resin 4.0's fixed size m-node structure for the distributed cache

E sales@caucho.com T 858.456.0300
W http://caucho.com F 858.777.3636

performance, the structure of the cache can be viewed as a collection of m-nodes.

Each server in a cluster maintains a collection of m-nodes, but only one triad server "owns" the management of any individual m-node. If a triad server owns an m-node, it is considered to have the authoritative, most recent version. The key hash determines the triad owner so finding an m-node is easy and unambiguous. While there is only one owner, the m-nodes are replicated on all the triad servers for redundancy in the case of failure or maintenance. Cache data is distributed in a similar way, except that because there is no version associated with the data value itself (versions are only a part of the m-node), no synchronization is necessary, only replication.

Applications running on Resin 4.0 may use the cache for a variety of reasons such as to cache an object or store session data. When using one of these facilities, the Resin 4.0 server that

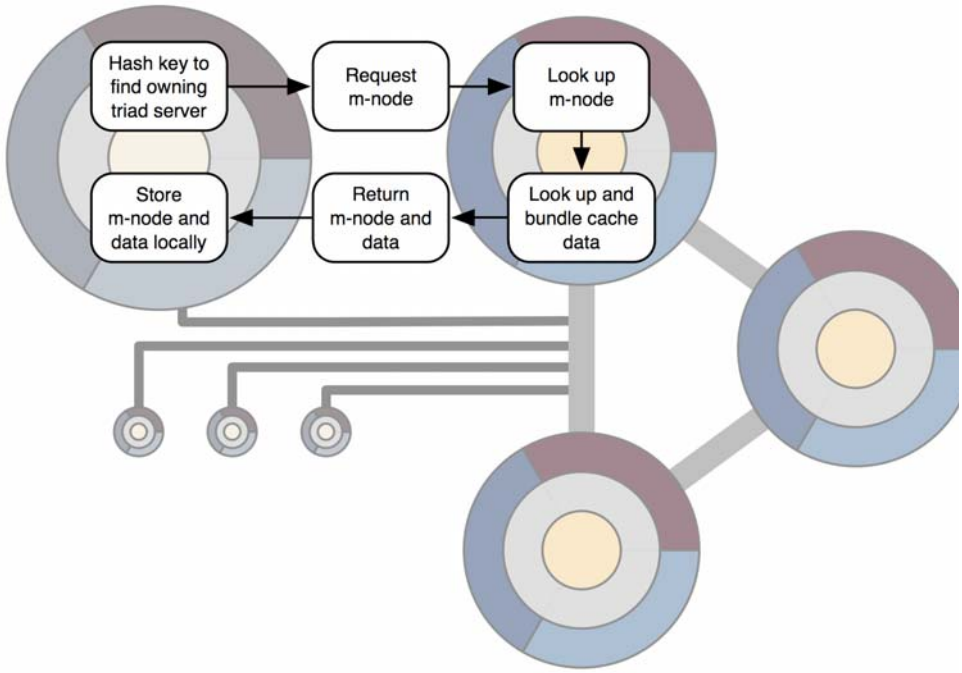


An application running on a dynamic server updates a cache entry by contacting the triad servers.

E sales@caucho.com T 858.456.0300
W http://caucho.com F 858.777.3636

received the web request interacts with the distributed cache by communicating with the triad servers in the cluster. To update a value in the cache, the server hashes the key using the SHA-256 algorithm. The server computes the m-node's owner using this key hash, then sends the new m-node to that triad server. Similarly, the server hashes the cache data to find its owner and sends the data itself to its owning triad server. The m-node and the data may be owned by the same triad server or separate ones. Once the server or servers receive the update, they transmit the data to the other triad servers for redundancy. The triad servers persist the cache data to a database so that recovery from transient errors is fast. The server that is performing the update does not have to wait for the triad to replicate or persistent the data; this process is performed asynchronously. The process to add a new cache entry is identical.

When the application tries to obtain a value from the cache, the server first hashes the key that the application requested. The key hash determines the triad server which owns the m-node. The server then contacts the owner triad server to request the m-node. The triad server looks up the m-node and inspects the value hash to find the actual data. Because the data is replicated on all the servers, the triad server that owns the m-node also has a copy of the value data. When the m-node owner server responds to the requesting server, it includes both the m-node and the associated data.



An application on a dynamic server retrieves a cache entry for the first time by contacting the triad.

E sales@caucho.com T 858.456.0300
W http://caucho.com F 858.777.3636

Customizing Cache Configuration

Multiple caches can be configured within each server for different applications and different requirements for each cache. Specifically, each cache has a variety of timeout values and other configuration options that customize the behavior for individual applications. For example, an application might cache certain computations based on a database query such as a list of current events. The application would like the list to be updated at least every minute to make sure that new events are shown to the users. Setting the expire timeout to 1 minute would achieve that goal.

Timeout	Description	Default
Expire timeout	The maximum time without an update that an item is considered valid.	Infinite
Local read timeout	How long a value is used locally before checking with the triad for updates.	10ms
Idle timeout	The maximum time without an update or a read that an item is considered valid. Typically used for sessions.	Infinite
Lease timeout	If items are leased with this cache, this is how long each lease is issued. Typically used for sessions.	5min

Configuration options for various distributed cache timeouts. These values apply to all entries contained within a particular cache.

HTTP sessions are another interface to the distributed cache. Specifically, session data can be stored in the cache under a key using the session id. HTTP session data has different requirements than most object cache data however. One requirement is that sessions be able to expire after a certain amount of time. However in this case, a session is considered live not only when it is updated, but also when it is simply accessed. By setting an idle timeout on the cache storing the session data, the sessions will timeout properly.

Sessions are used by applications in a very predictable way, so optimizations can be made in the network infrastructure and the cache to improve performance. If the user interacts with the application using the same server for the duration of the session, that session data can be kept

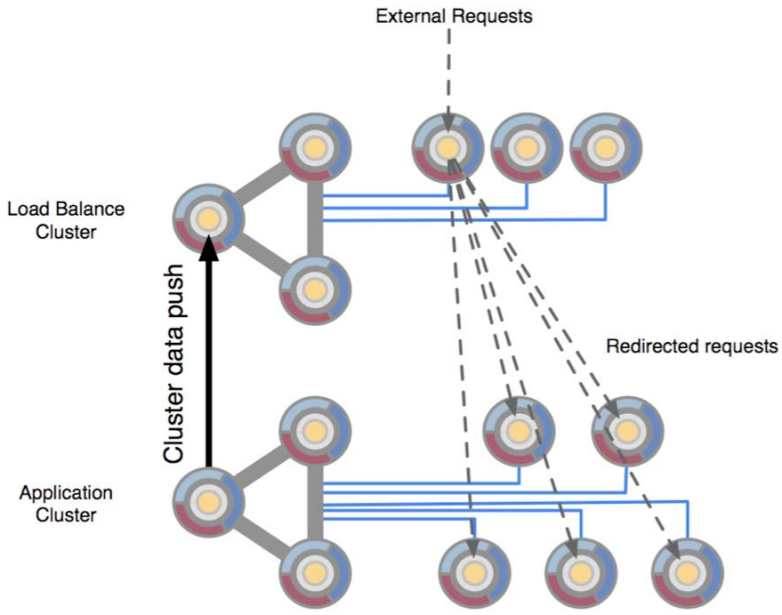
E sales@caucho.com T 858.456.0300
W http://caucho.com F 858.777.3636

in local memory for fast access. Load balancers can force this behavior by implementing **sticky sessions**, a way of routing requests from the same user to the same server during a single session. Resin 4.0's load balancer implements sticky sessions for this reason.

The cache can also be optimized to take advantage of sticky sessions. In the normal case that a single server is the only one to read and write a session, it can be given a non-exclusive **lease** on the session data. The session itself is stored as a cache entry, so the triad member that owns the entry's m-node will issue a timed lease on the m-node to the server handling the first request in the session. Until the lease expires, that server can assume that its local copy of the cache is up to date and therefore never needs to use the network to check the validity of its copy with the triad. To maintain reliability, the lease-holding server still writes all of its changes back to the triad. Other servers may also access and update the session. If any changes are made by any server other than the lease holder, those changes are sent to the lease-holding server by the triad as they are made. Thus in the normal case, the access to session data is much faster, but reliability and fail-over are maintained if a server fails.

Resin 4.0 Load Balancer

Resin 4.0 includes a software load balancer that can distribute requests to servers in a cluster. This load balancer is simply a Java EE application, so it runs in its own separate cluster, potentially on a number of servers. All Resin 4.0 clusters maintain a list of the dynamic servers available in a special distributed cache entry. The load balancer cluster acts as a read-only client of the application cluster's distributed cache, with the application cluster sending updates to the load balancer cluster each time a dynamic server is added or



An example load balancer scenario with two tiers and multiple load balancers. The current state of the application cluster is pushed to the load balance cluster by the application cluster's triad.

E sales@caucho.com T 858.456.0300
W http://caucho.com F 858.777.3636

removed. When one of the load balancer servers distributes requests, it simply looks in the distributed cache to see what application cluster servers are available.

The load balancer can distribute requests using a number of algorithms:

1. Round robin
2. Server load
3. Green load balancing

Round robin is the simplest algorithm for load balancing in which the next request is sent to the next server on the list. The triad servers maintain the list of servers and their order, so if new servers are added or removed the load balancer will follow that order.
The server load algorithm uses a set of empirical data to determine which server in the cluster is the least loaded, then assigns the next request to that server. Specifically, the number of active connections that the server is handling at the moment along with the CPU load of the server are included in a formula to calculate the server load. In addition to these measured values, servers may also be weighted explicitly to direct more or less load. This algorithm also takes into account startup time for a server and allows it time to “warm up” so that the applications are ready to handle requests.

Green load balancing uses the same measurements as the server load algorithm, except that instead of directing the next request to the least loaded server in the cluster, this algorithm tries to load a single server up to a threshold level before moving to the next server. This approach means that only the number of servers that are needed to handle the current load will be used. Thus servers that are not necessary during periods of low load can be placed in low power mode to preserve energy and reduce wear.

Accessing the Distributed Cache from PHP applications

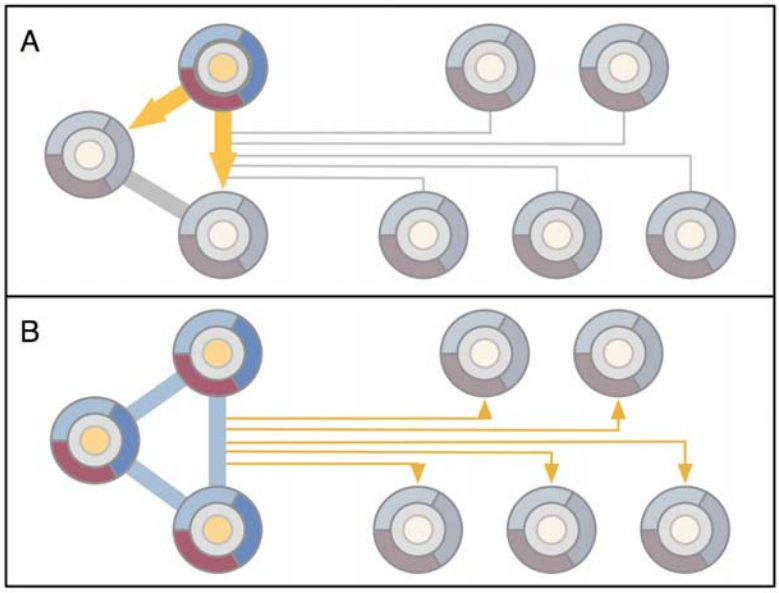
Resin 4.0 includes Caucho Technology's implementation of PHP called Quercus. This implementation is written in pure Java and runs within the JavaEE Servlet framework. PHP pages are executed in a fashion similar to JSPs. PHP contains library functions to implement HTTP sessions as well as object caching. By implementing these functions using Resin 4.0's distributed cache, PHP applications can also take advantage of the benefits of the architecture. Moreover, PHP applications can share cached data with Java applications running within the same cluster.

E sales@caucho.com T 858.456.0300
W http://caucho.com F 858.777.3636

The session API in PHP is straightforward and backed by the distributed cache much in the same style of Java Servlet sessions. PHP also includes an API called APC to provide object caching which is used widely in both custom and open source PHP applications such as MediaWiki. APC provides a basic key-value storage mechanism with timeouts much like the Java Cache API. These functions are also implemented in Quercus using the Resin 4.0 distributed cache directly.

IV. Cloud-wide Application Replication and Deployment

Administrators and developers can deploy applications to the cluster using a number of tools supported by Resin 4.0 such as ant, Maven, or Eclipse. These deployment tools distribute a Web Archive file (.WAR) to all of the cluster, avoiding the need to write tedious scripts. The applications are deployed using a transactional protocol, so if the application is not received successfully by a server in the cluster, it will not start.



Once an application is deployed to one of the triad members, it is replicated across the other two (A). When all of the triad servers have the updated application, the data is pushed out to dynamic servers (B).

Then when deploying an application, these tools contact one of the cluster triad members which will then distribute the contents of the web application to the other two triad members. When all of the triad members have replicated the application files, they update the dynamic servers by pushing out the new data. Once all of the dynamic servers are updated, the new application can be started either automatically or manually.

If the application is updated, the administrator can then repeat this process. However because most application updates include many of the same files, Resin 4.0 sends only the different

E sales@caucho.com T 858.456.0300
W http://caucho.com F 858.777.3636

files to the other triad and dynamic servers. This approach is known as an **incremental update.** When a new dynamic server is added to the cluster, it contacts the triad and downloads all the applications currently deployed to the cluster.

This deployment mechanism also allows for graceful transitions to new versions of applications as well. By **enabling versioned web application deployment**, a newly deployed version of the application will only serve new sessions. All existing sessions will continue to be served by the previous version of the application. Only when a session expires or is explicitly invalidated will an active user be directed to the new version.

Distributed Git Repository

The application files that an administrator sends to the triad are stored persistently in a distributed Git repository. The Git repository format was developed as a fast, open source, distributed version control system by Junio Hamano and Linus Torvalds. Resin 4.0 uses the Git repository format on each triad and cluster server to store application and configuration files. The unique properties of the Git repository provide for some benefits such as:

- **Transactional updates**

  If an update does not succeed or is interrupted, the new application or configuration files are not distributed to either triad members or dynamic servers. Only when the files are verified to be correct on each server are they made live.

- **Highly concurrency**

  The Git repository format ensures that data is written in isolation, but accessible by any number of readers. Thus when updating application files, the chance of corrupting files by multiple writers is virtually zero, resulting in faster performance.

- **Durability**

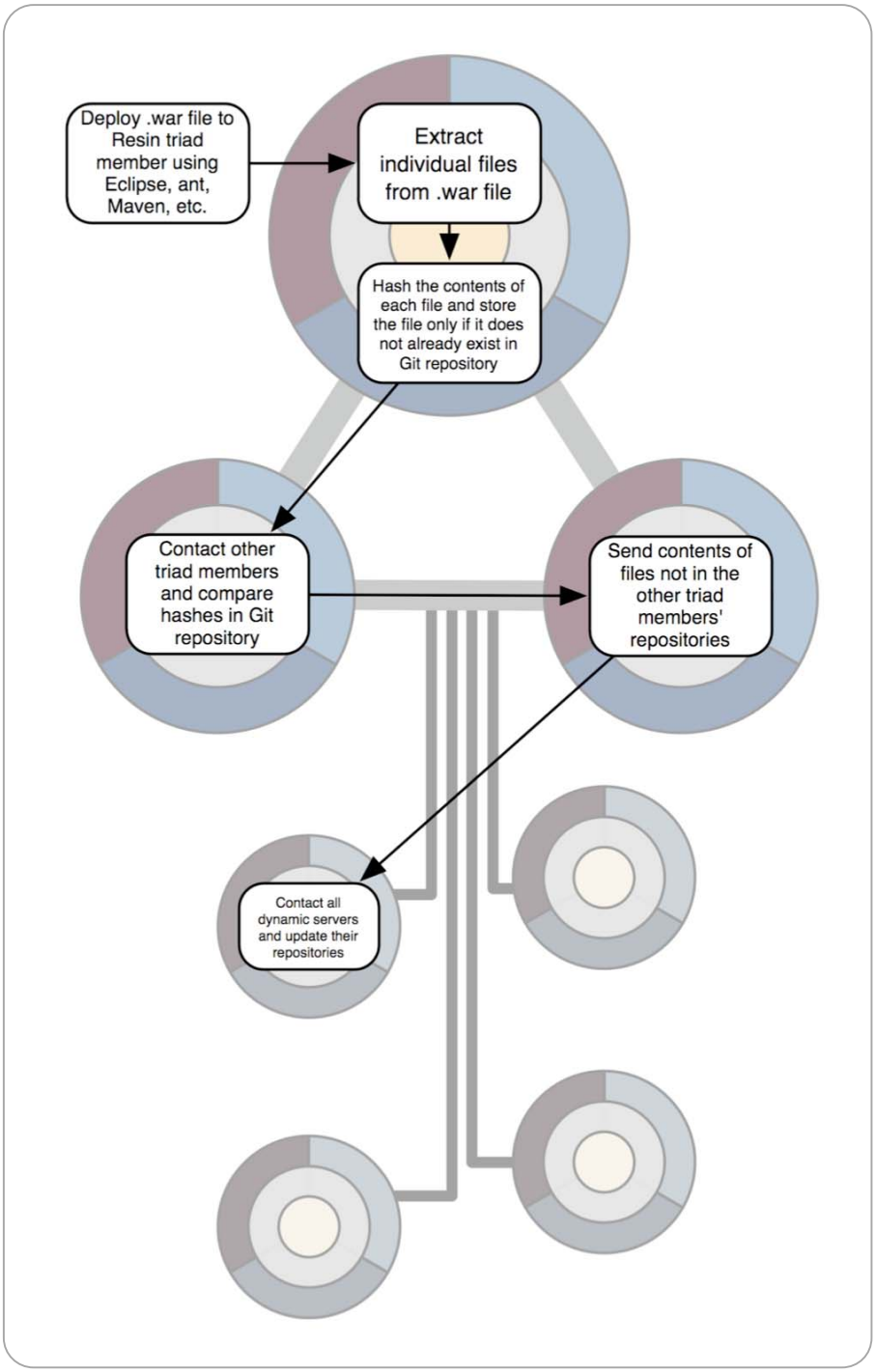
  By writing application and configuration files to disk on all three triad servers as well as the dynamic servers, the application can survive numerous failures in the cluster.

- **Incremental updates**

  By examining all of the files in an application individually, Resin 4.0 uses the version control features of the Git repository so that only new files are sent over the network, thus reducing deployment time, network traffic, and storage requirements.

E sales@caucho.com T 858.456.0300
W http://caucho.com F 858.777.3636

The Git repository uses secure hashing (SHA) to store files according to their contents. When a new file is added to the repository, its contents are hashed and compared to the hashes of files already stored in the system. If this is the first time the contents of the file have been stored in the repository, a new entry is made and the file is stored. If a file with the same contents has been stored in the repository before, no additional space will be allocated. When directories are stored in the repository, they are stored as trees whose hash is the contents of the directory. Thus file names are not used directly.



This approach means that if any two attempts are made to store two files with different contents under the same name, they cannot conflict. Thus there is no danger of race conditions to write the same file twice. When the files are read, their contents are also checked against the value of the hash under which they were indexed. By using this algorithm, no partial or corrupted values can be written into the repository either. Finally, the hash-based storage of files means that any files that are unchanged between two versions of an application are not stored twice, nor do they require retransmission over the network.

E sales@caucho.com T 858.456.0300
W http://caucho.com F 858.777.3636

Using this repository structure, Resin 4.0 replicates application files across the triad and dynamic servers. When a web application archive (.WAR file) is deployed to the system, the triad member that receives it will examine the files contained within and store them individually into the Git repository. If these files are a new version of an existing application, only the new and updated files will be stored. When the triad member contacts other triad members or a dynamic server, they compare only the hash values in their repositories first. Only files whose hash values do not exist in the server to be updated are sent over the network.

V. Use cases

i. Setting up a basic Resin 4.0 cluster

A cluster containing a single pod is the easiest to set up for an application. For many organizations this scenario will be sufficient to deploy, run, and scale their applications. Using a single pod is useful in more traditional situations where the organization controls all of the deployment servers, but wants the flexibility to add and remove servers without downtime or reconfiguration of the application server.

The configuration of the cluster starts by explicitly identifying the triad servers in the Resin 4.0 configuration file. This file is then installed manually on the triad servers and Resin 4.0 servers can then be started. At this point, the triad is now able to accept new dynamic servers into the cluster.

To add a new server, the administrator first logs into one of the triad members to register the new server. The administrator may use either a JMX-based tool or the Resin 4.0 web-based administration console. Next the administrator starts a new virtual machine. On start up of this virtual machine, the operating system starts an instance of Resin 4.0, passing the address of one of the triad servers via command line options. The new dynamic server contacts the triad server to join the cluster and download its configuration and applications. Because this server was registered with the triad earlier, the triad is expecting the new cluster member. Once the dynamic server authenticates itself, the triad will then send the configuration and application files. So far no application files have been deployed, so only the configuration will be sent at this time.

Next the administrator deploys an application to the triad using either ant, Maven, or Eclipse. The deployment tool adds the application to the triad's Git repositories and the triad then pushes it to any dynamic servers that are already part of the cluster. Should any new dynamic

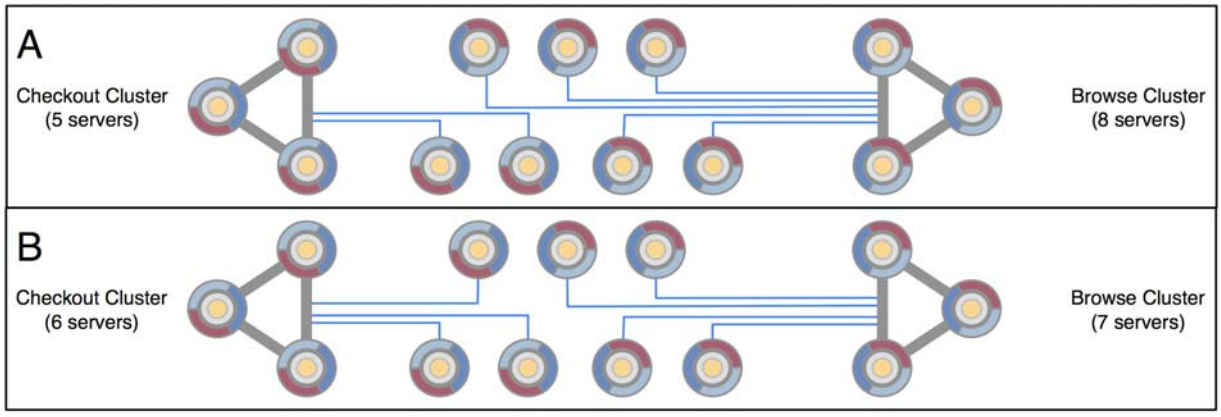
E sales@caucho.com T 858.456.0300
W http://caucho.com F 858.777.3636

servers join after the applications are deployed, they will receive the files when they register with the triad. Updates to the application files and new application versions will also be pushed out to any dynamic servers that are part of the cluster when they are deployed.

ii. Dynamically Reprovisioning Resources among Applications using Multiple Clusters

Each cluster in a Resin 4.0 deployment should serve the same set of applications. For example, consider a commerce site that offers a searchable online catalog of goods and a shopping cart to check out and pay for the goods. In this example, a single cluster may serve all of the applications related to checking out and making payments. The applications which deal with browsing the site and searching are grouped together in a separate cluster.

Depending on the current load of the applications and the needs of the users, different applications may need different levels of resources. For example, say that during a normal week, most customers browse the catalog and only a small portion of those visiting the site end up making a purchase. The administrator may decide to dedicate a cluster of 8 machines to the browsing applications, but commit only 5 servers to checkout. During a sale however, the checkout traffic may grow much greater in comparison to the browsing traffic. Based on the load seen by the servers, the administrator may choose to allocate 6 servers to checkout and 7 servers to browsing.



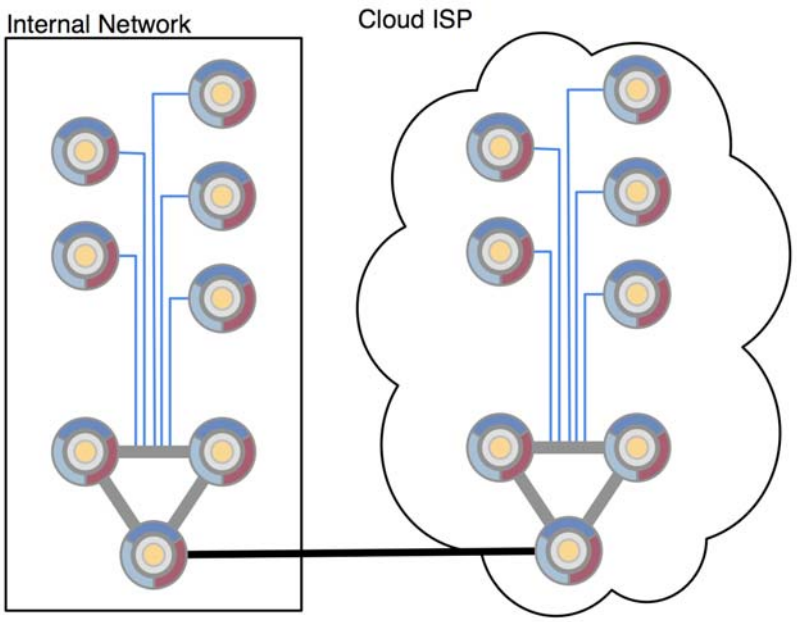
An example of reallocating resources from one cluster to another. Normally, the checkout cluster load is smaller so the administrator only allocates 5 servers (A). During a sale, the administrator moves a server from the browsing cluster to the checkout cluster (B).

E sales@caucho.com T 858.456.0300
W http://caucho.com F 858.777.3636

To reallocate the resources, the administrator would shut down 1 of the dynamic servers in the browsing cluster. The other servers in the cluster remain available and can take over the sessions that were being served by that dynamic server because the triad has copies of all the session data. The administrator then starts a new dynamic Resin 4.0 server in place of the ones just shut down, but now adds it to the checkout cluster. The checkout cluster's triad sends the application files so that the dynamic servers can now handle additional checkout traffic. Using virtual machines that are preconfigured to act as either checkout cluster servers or browsing cluster servers can make the process even easier.

iii. Elastic cloud ISP

A new industry of elastic cloud ISPs is emerging in which an organization can run a virtual machine on the ISP's hardware at a metered rate. The administrator of an organization creates a virtual machine image which he or she then uploads to the ISP and starts via a web console or web service. Resin 4.0 can take advantage of this type of infrastructure to give organizations the ability to add new server capacity at will. Many organizations have only seasonal needs for additional computing capacity due to sales, deadlines, or other scheduled events. These organizations can maintain smaller fixed server pools for their normal traffic level, but use the ISPs for the planned periods of higher traffic. Alternatively, certain these ISPs allow flexible capacity for unplanned traffic spikes



A single Resin 4.0 cluster deployed both to an internal network and a cloud ISP. Each network contains one pod of the cluster and the pods are connected to allow sharing of cache and application file data.

E sales@caucho.com T 858.456.0300
W http://caucho.com F 858.777.3636

as well.

Take as an example the case where an organization has a set of servers that it maintains full time, but wants to add capacity during high traffic events. The administrator would configure a Resin 4.0 cluster with two pods, one for the internal servers and one to be run on the ISP. The internal pod runs full time, but the ISP pod may remain shut off or have few dynamic servers most of the time. When a high traffic event occurs, the administrator starts the ISP pod and adds new dynamic servers to it to allow for additional capacity. The servers in the each pod only have to consult their triad for most operations, leading to fast access to the cache and sessions. To ensure security, cluster pods can be configured to send encrypted and signed messages when in an environment such as a cloud ISP.

VI. Conclusion

Resin 4.0 adds a number of features to enable cloud computing for web applications written in Java and PHP. While this paper describes the mechanisms behind these features such as distributed caching and cluster-wide application deployment, the developer does not have to tailor any code to Resin 4.0 and may continue to create standard JavaEE or PHP applications. These features simply improve the capacity, reliability, and availability of those applications. Cluster-wide application deployment and dynamic clustering make the task of maintaining virtualized deployments much simpler for administrators by providing easy deployment and scaling tools. By offering these features, Resin 4.0 provides a web application platform to exploit the full capabilities of cloud computing.

About Caucho Technology

Caucho Technology is an engineering company devoted to reliable open source and high performance Java-PHP solutions. Caucho is a Sun Microsystems licensee whose products include Resin application server, Hessian web services and Quercus Java-PHP solutions. Caucho Technology was founded in 1998 and is based in La Jolla, California. For more information on Caucho Technology, please visit www.caucho.com.

E sales@caucho.com T 858.456.0300
W http://caucho.com F 858.777.3636