The PHP Company

White Paper:

Session Clustering in PHP

How to Implement a Scalable Failover Solution for PHP Sessions

By Shahar Evron,
Technical Product Manager, Zend Technologies LTD.

June 2010

© 2010 Zend Technologies, Ltd. All rights reserved.

Table of Contents

Background: PHP Sessions **3**
State Representation in HTTP 3
HTTP Cookies 3
PHP Sessions 3
Existing Session Storage Engines 4
Files Storage 5
In-Memory and other local storage 5
Relational Database Storage 5
Distributed Cache Storage 6
Zend Session Clustering 7

Introducing Zend Session Clustering **8**
Architecture Overview 8
Disk vs. Memory Storage 8
Redundant Session Storage 9
Normal Operation Workflow 10
Handling Server Faliures 11
Scaling Up 11
Scaling Down 12
Graceful Startup 12

Appendix: Tuning Session Clustering **13**
Tuning Session Clustering for Performance 13
Other Important Configuration Options 13

Background: PHP Sessions

State Representation in HTTP

HTTP, the protocol over which the web is built, is a stateless protocol. Each HTTP request is user session context independent, and the server is, on the HTTP protocol level, unaware of any relationship between consecutive requests.

This has made HTTP a highly scalable and versatile protocol. However, in most Web applications some notion of a user session – that is short-term, user specific data storage, is required.

For example, without some sort of state representation a web application cannot distinguish between logged-in users (or technically put, requests coming from an HTTP client that has logged in) and non logged-in users. In many cases even more complex data, such as the contents of a shopping cart, must be maintained between requests and attached to a specific user or browser.

HTTP leaves the solution of such problems to the application. In the PHP world, as in most web-oriented platforms, two main standard methods exist for storing short-term user specific data: Cookies and Sessions.

HTTP Cookies

Cookies are set by the server, and are stored by the browser on the end user's machine. Browsers will re-send a Cookie to the same domain in which it originated, until it expires. This allows storing limited amounts of user-specific information and making them available to the application on each request made by the same user. Cookies are convenient and scalable (no storage is required on the server side), but are also limited due to a number of reasons:

- Cookies are limited in size, and the limit varies per browser. Even if the limit is high, large amounts of data sent back and forth in each request may have a negative effect on performance and bandwidth consumption.
- Cookies are sent repeatedly, on each request to the server. This means that any sensitive data contained in cookies is exposed to sniffing attacks, unless HTTPS is constantly used – which is in most cases not an effective option.
- Cookies store strings – storing other, more complex types of information will require serialization and de-serialization to be handled in the application level.
- Cookie data is stored on the client side – and as such, is exposed to manipulation and forgery by end users, and cannot be trusted by the server.

These limitations make it almost impossible to rely on Cookies to solve all state representation problems. As a better solution, PHP offers the Sessions concept.

PHP Sessions

When a PHP user session is started, the user is assigned a random, unique *Session ID* that is sent by the server to the browser. This session ID is then used as the key to a storage area on the server, which is unique for this session.

The Session ID is passed back and forth from the client to the server using either Cookies (as is more common in recent years) or by appending the Session ID to the URL's query string. The only information sent between the client and the server is the ID – usually a random, short string of characters. Session IDs are random and hard to guess, and are meaningless without the actual data stored on the server – and thus forging them is pointless.

On the server side, session information can be stored in many ways. PHP offers a modular session mechanism, with a simple API for session handling, but with an ability to easily switch the storage backend by changing the value of a configuration option, without any code changes.

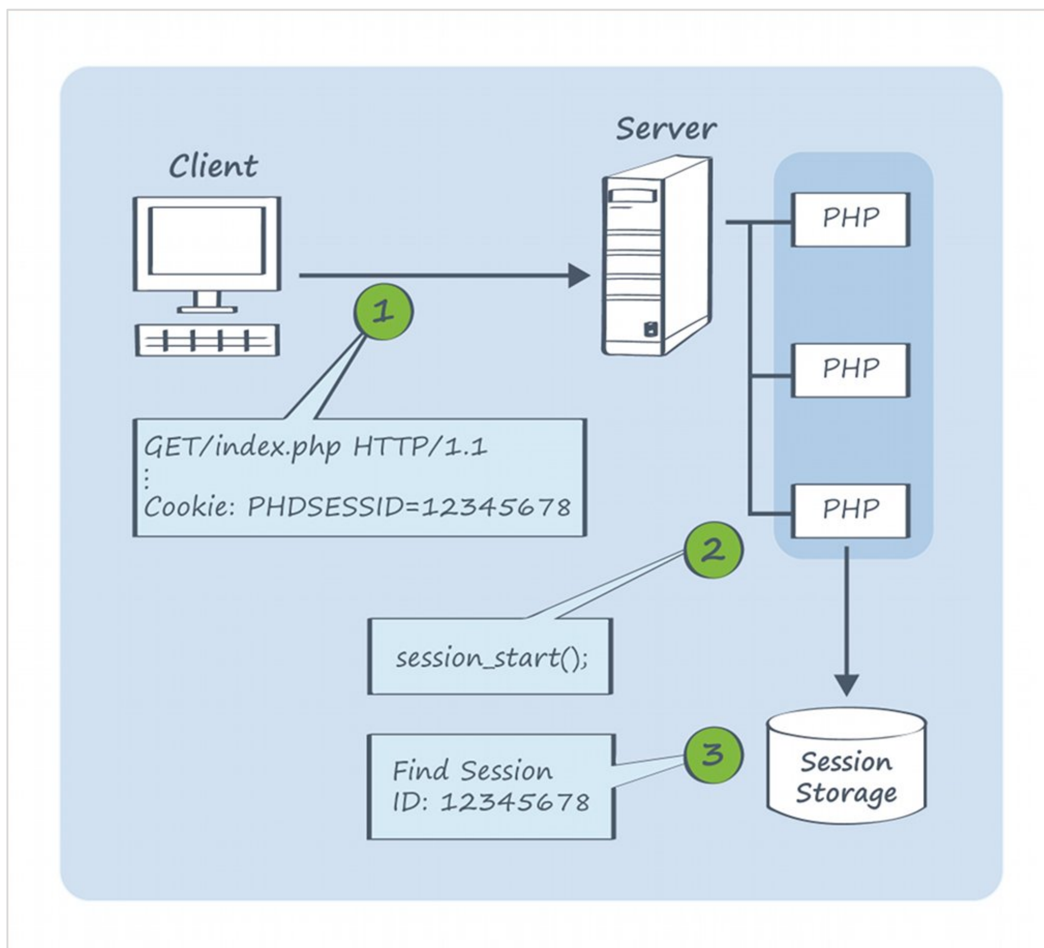

Figure 1 – Basic session handling flow in PHP. The Browser sends a unique ***session ID*** *(1) on each request, using a Cookie or as a GET/POST parameter. When* ***the session_start()*** *PHP function is called (2), PHP asks the* ***session save handler*** *for the session identified by the* ***Session ID****. It is then up to the storage engine to find this session (3) and return it back to the PHP.*

By default, PHP stores session data in files on disk – this storage method works well and does not depend on any external system, as long as requests coming from a single user will always end up on the same server with access to the saved session files. As applications grow, this usually becomes impossible or impractical – and then other session storage backends (or "save handlers" as they are sometimes referred to) should be used.

Existing Session Storage Engines

There is a multitude of session save handlers available for PHP, each with its own advantages and disadvantages. Given the modular nature of session storage in PHP, users are free to create additional save handlers, either as C/C++ extensions, or by using user-space PHP code.

The following list covers some of the more widely used session handlers, and overviews their capabilities:

Files Storage

The '*files'* save handler is PHP's default save handler. It stores session data as files on disk, in a specified directory (or sometimes in a tree of directories). Usually, this save handler is capable of easily handling tens of thousands of sessions.

The biggest disadvantage of the *files* save handler is that it imposes a major scalability bottleneck when trying to scale an application to run on more than one server.

In most clusters, requests from the same users may end up on any of the servers in the cluster. If the session information only exists locally on one of the servers, requests ending on other servers will simply create a new session for the same user, resulting in data loss or inconsistency.

One way to work around this problem may be to store session data on a shared file system such as NFS (*Network File System*). Unfortunately, this is a highly inefficient method and usually results in performance problems and data corruption. This is due to the fact that NFS was not designed for the high read/write ratio and potential concurrency levels required for session handling.

Another potential solution is to use a "sticky", session aware load balancer. Most load balancers today have stickiness capabilities in some form or another. While this is a better solution, experience shows that in high loads sticky load balancers tend to become a bottleneck, and cause uneven load distribution. Indeed, most heavy PHP users prefer to use round-robin load balancing with other session storage mechanisms.

In addition, sticky load balancing does not solve another inherit disadvantage of the *files* session handler: session redundancy. Many users rely on clustering for application high-availability. However, in many cases session high-availability is also important. If a server crashes or otherwise becomes unavailable, the load balancer will route the request to a different server. The application in this case will still be available – but the session data will be lost, which in many cases may lead to business loss.

In-Memory and other local storage

Several PHP extensions offer session storage capabilities in the server's RAM – these include the *mm* module and *eaccelerator*.

While these storage methods may (or may not) provide better performance than the traditional *files* handler, they suffer from the same inherent problems of scalability and high-availability. In addition, local memory storage solutions tend to be even less fault-tolerant due to their volatile nature in case of memory corruptions or crashes due to software bugs.

PHP's *sqlite* extension also provides a session storage handler, which stores sessions in a local SQLite database file. Again, the same disadvantages apply here as with other local-only save handlers.

Relational Database Storage

Several extensions and code libraries (including Zend Framework) offer session save handlers that store session data in a relational database such as MySQL or Oracle.

Session storage in a central database solves the scalability limitation imposed by local storage mechanisms, by making the session data available to all servers in the cluster, and making sure data integrity is maintained.

Database session handlers work more or less the same, regardless of implementation and database used for storage; one or more tables are created using the session ID as the primary key, and with another column for serialized session data. With each new session, another row is inserted to this table. On each request with the same session ID, this row is fetched in the beginning of the request and is then updated at the end of it.

This storage method is obviously more scalable than local storage, but still has several disadvantages which should be pointed out:

Session Storage is very different from regular data that is usually stored by web applications in a relational database. Sessions have an almost 1:1 read/write ratio (PHP will save the session at the end of each request that opened the session, even if the data did not change), and as such, row-level locking is required to maintain session integrity. Database-resident query caching usually does not work well with such usage patterns. To gain optimal performance, the session storage database should be isolated from the application's database, and tuned differently. This imposes an additional maintenance overhead, at least in higher load levels.

When a single database is used for an entire cluster, the single database quickly becomes both a performance bottleneck and a potential point of failure and subsequently, database clustering is required. Again, this causes a maintenance overhead whenever the application scales up.

Distributed Cache Storage

Another possibility for storing cluster-wide session information is using distributed caching systems, namely *memcached*.

Memcached[i] (pronounced "mem-cache-D") is an open-source distributed memory based storage system, designed mostly as a fast caching solution. In recent years it has gained much popularity as a distributed caching system for web applications, and can also be used for session storage in PHP.

Session Storage in memcached solves the locality problem, and is usually faster than most DB session storage implementations. Memcached is distributed in the sense that data may be spread on multiple nodes – this makes memcached scalable when it comes to handling a growing number of concurrent sessions.

However, it is important to understand that memcached is in fact a large, distributed hash-table: each data item (or session in the case of session storage) is stored in one location on a single server. The server to connect to in order to fetch a data item is determined on the client-side by the key, in our case the session ID.

This architecture works extremely well as a caching system, however for session storage it imposes several limitations:

- Data Redundancy – while memcached could be clustered and even if one memcached node becomes unavailable the application will continue to function, any sessions stored on that node will be lost. When used for caching, this is not a real limitation, because data can always be restored from its original location. However, in some applications this potential for session loss is not acceptable.
- Scaling Up is not seamless – as applications grow and require more space for session handling, you might need to add additional memcached nodes. Adding a memcached node requires re-configuration of all PHP servers in the cluster. By itself this is only a small headache, but because memcached clients (PHP in this case) choose the node to connect to in order to search for a particular data item based on the key and the list of memcached nodes they are aware of, such reconfiguration may mean multiple session loss. Again, as a caching system this is usually acceptable, but may not be an option for session storage.
- Cyclic memory – when memcached runs out of its memory limit, it will delete old items. When storing many sessions, especially with long session lifetime, this may lead to session loss.

Zend Session Clustering

Zend Session Clustering is a part of Zend Server and Zend Server Cluster Manager – Zend's enterprise-grade commercial PHP web application server. Session Clustering is installed on Zend Server instances and is managed and configured by Zend Server Cluster Manager.

On systems managed by Zend Server Cluster Manager, the *cluster* save handler is made available to PHP. When enabled, sessions are shared across the cluster transparently, like with other save handlers.

In comparison to other session storage solutions, Session Clustering offers several benefits, including linear scalability, high-availability of session data, high performance and utilization of existing infrastructure.

The following chapter explores Zend Session Clustering in greater detail.

Introducing Zend Session Clustering

Initially introduced in version 2.0 of Zend Platform back in 2005, and with multiple enhancements and new capabilities added over the years, the release of Zend Server Cluster Manager makes Zend Session Clustering available to Zend Server users.

Zend Session Clustering is a PHP session storage mechanism which uses a network of independent daemons running on frontal web servers to store and share sessions across the cluster.

Session Clustering is designed to provide a fault-tolerant, high-performing session storage mechanism for PHP clusters, while maintaining linear scalability and ease of setup. It allows maximal re-use of existing hardware, and does not require any additional servers beyond the ones used as frontal web servers. If you have a sticky or semi-sticky load balancer, Session Clustering will work well with it, and in fact will perform even better with higher levels of session affinity. When a server in the cluster fails, or when the load balancer fails to maintain session affinity, the session data will still be available.

Switching from other session save handlers to Session Clustering is usually a matter of changing a configuration directive. In the vast majority of cases, no code changes are required[ii].

Architecture Overview

From a high level perspective, Zend Session Clustering is composed of 3 main parts:

- The **Session Clustering Extension** (or "*mod_cluster*") is a PHP extension enabling the use of Session Clustering as a session save handler. The main role of this extension is to communicate with the locally installed Session Clustering Daemon.
- The **Session Clustering Daemon** (sometimes abbreviated "SCD") is an external service that runs on each PHP server in the cluster. This daemon is the heart and brain of the Session Clustering system, and is responsible for managing PHP sessions and sharing them across the cluster. The daemons are also responsible for maintaining session redundancy by assigning backup servers for each session. SC Daemons on the cluster communicate with each other directly, without relying on a single management service.
- The **Storage Backend** is a part of the Session Clustering Daemon and this is where sessions are actually stored. Session Clustering provides two storage backends: The first is memory based (default), and the second is disk based. Each storage backend has its pros and cons, and neither requires any attention from the end user – storage backends are an integral part of the Session Clustering Daemon and require no setup or management.

The Session Clustering Extension communicates with the local SCD using a UNIX domain socket where available, or a local TCP connection. The PHP extension does not communicate with any other daemons in the cluster.

Session Clustering daemons communicate with each other over TCP for large data transfers using daemon-to-daemon connections, and UDP for sending short control messages to multiple daemons. UDP Broadcast or Unicast may be used, depending on configuration.

Disk vs. Memory Storage

By default, Session Clustering stores sessions in memory. However, on-disk storage is also possible.

Memory storage is, in most cases, faster – this is the main benefit of memory session storage.

Disk storage on the other hand is slightly slower, but is preferable when session data is exceptionally large - that is when storing tens of thousands of sessions may take up more memory than is available on the server.

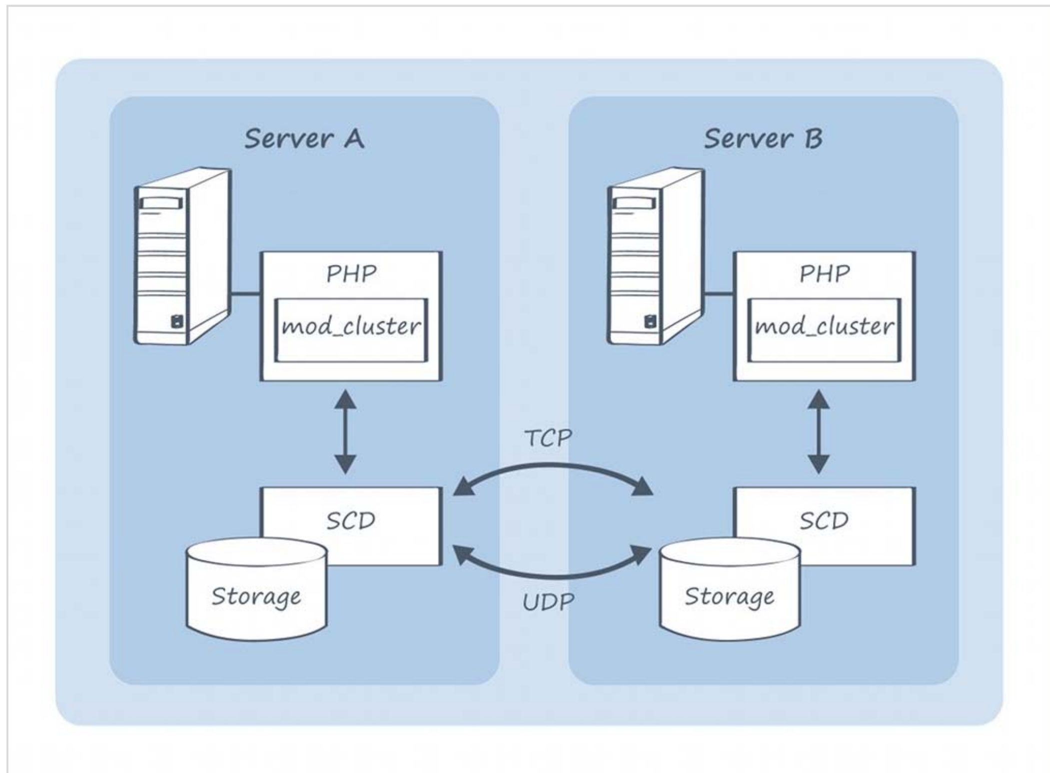

Figure 2 – The high-level Session Clustering Architecture, including mod_cluster (1), the Session Clustering Daemon (2) and the storage backend (3).

Redundant Session Storage

Each new session is stored in two locations: on a *master* server and a *backup* server.

When a new session is created, the master server will be the server on which the request to start a new session was received. The SCD on that server will create a new session, and will find a backup server by picking the least-loaded server in the cluster.

The internal identification of both the master and backup server is encoded into the Session ID given to the user. This allows each server in the cluster to find both the master and the backup servers by simply looking at the Session ID token provided by the client, and there is no need to perform any hash-table lookups or querying in some central database.

```
      Server  Apache/2.2.9 (Debian) PHP/5.2.13
  Set-Cookie  PHPSESSID=b05fcb78-baba97f6-b05e5b7b-baba97f6-00000002-j9k75h8905vqq2da6s9ckffv86;
        Vary  Accept-Encoding
X-Powered-By  PHP/5.2.13 ZendServer/5.0
```

Figure 3 – The "Set-Cookie" HTTP header sent back by the server, setting a Session Clustering PHP Session ID. In Session Clustering, the Session ID format is composed of encoded identifiers of the master server (first 16 characters) and backup server (next 16 characters), a revision number used for conflict resolution (next 8 digits), and a random, unique ID (last 32 characters).

This replication of each session to two servers provides a very good balance between performance and high-availability. While replication to more servers could guarantee a better chance of session availability even in cases of multiple server failures, the performance

overhead of such replication is under most circumstances unacceptable, and scalability is problematic. In most clusters, the chance of multiple servers failing at the same time is very low.

Normal Operation Workflow

As an example, let's assume we have a 3 server cluster, in which the servers are named "Server A", "Server B" and "Server C".

When a user's first request is received by the cluster, it is randomly routed by the load balancer to one of the servers. As our PHP code calls the *session_start()* function, a new session is created for our user. If the request is handled by Server A, Server A will choose a backup server for the session, (say, Server B), and will set a new Session ID for the user, identifying Server A as the master and Server B as the backup encoded into it.

If Server A receives the next request as well, the server will know to look for the session in its local storage. Once the session is saved again, it will take care of updating the backup server (Server B) with the new session data.

But what if the next request ends up on Server C? In this case, Server C will figure out (by looking at the Session ID) that Server A is the owner of the session, and will connect[iii] to it and ask for the session data. Server C will not copy or take over the session as, sessions do not move from their original master and backup servers, unless one of them fails.

With Session Clustering, network access and session retrieval time are greatly reduced if a sticky load balancer is used. Since sticky load balancers typically direct users to the server in which their session was created, sessions are almost always fetched from local storage. Network access will only be needed in case of server or load balancer failure, in which case Session Cluster will ensure session fail-over.

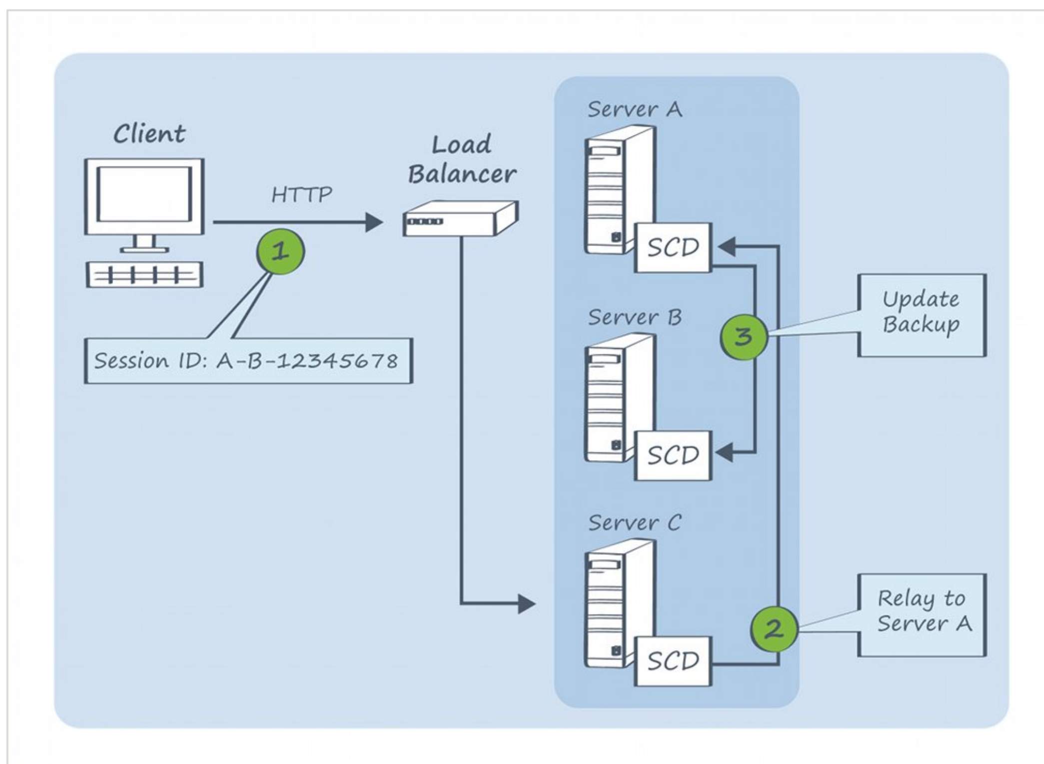

Figure 4 – Session sharing workflow. The client sends out a request, with the Session ID designating Server A as the master and Server B as backup (1). Server C picks up the request and asks Server A for the session (2). Once the request is complete, the session data is saved again by Server A, which takes care of updating the backup, Server B, as well (3).

Handling Server Failures

If Server C attempts to fetch a session from Server A but fails (for example if Server A crashed), it will reach out to the server designates as the backup server, in this case Server B. Server B will assign itself as the new master for this session, and will designate a new backup server. The Session ID will be automatically updated to reflect this change.

In a similar manner, if the backup server becomes unavailable, the master server will take care of finding a new backup for the session. Again, the session ID will be updated accordingly.

The Session ID is only updated when the client sends an HTTP request. For this reason, if a server crashes between requests, a session may only reside on one server until the next request is received. In this intermediate situation, session redundancy is temporarily unavailable.

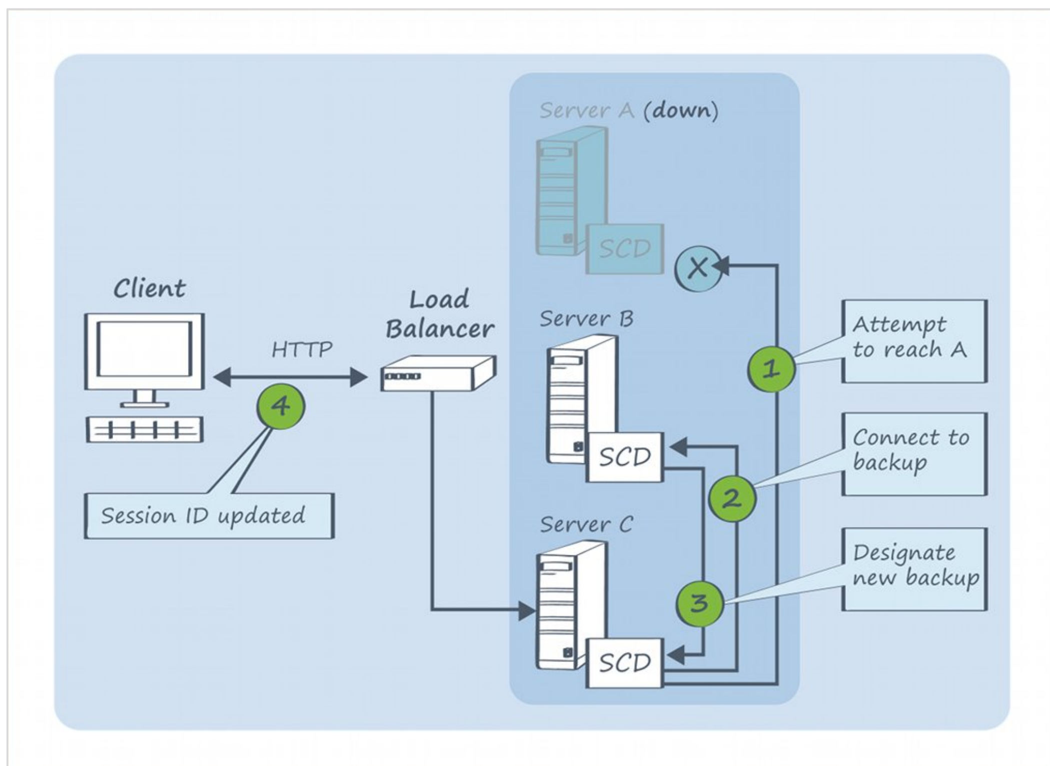

Figure 5 – If the master server becomes unavailable (1), the session is fetched from the backup server (2). The backup will then designate itself as master, and will find a new backup server (3). The Session ID is updated to reflect this change (4).

Scaling Up

In today's web environment, the ability to scale an application up as demand grows is usually a fundamental requirement. While Cloud infrastructure or arrays of commodity hardware allows for efficient hardware scalability, without the ability to scale the infrastructure supporting the application, such as the session storage mechanism, these capabilities are meaningless.

Session Clustering was designed with transparent, linear scalability in mind.

As you add new servers to the cluster, your capacity for linearly handling additional sessions grows. Zend Server Cluster Manager takes care of updating configuration on all cluster members, so that they are made aware of the new server. Traffic coming in to the new server will create new sessions, and the algorithm used for selecting backup servers will ensure that

session load distribution between all servers will eventually (and usually within minutes) become even.

Unlike some other session storage mechanisms, this happens transparently, without any need for restarts, session loss or hardware provisioning.

Scaling Down

With the introduction of Cloud Computing as a viable and increasingly popular deployment platform, the ability to scale an application down following a high-traffic period is more important than before. The ability to shut down servers when they are no longer required for current load levels, without losing existing sessions, can help reduce costs and maintenance overhead.

Session Clustering in Zend Server Cluster Manager provides a Graceful Shutdown mechanism. When a server is removed from the cluster (Server A in our example), the Graceful Shutdown process kicks in. The server will find and designate one of the other members of the cluster as a replacement server (Server B), and will transfer all active sessions to that server. During this time Server A will no longer accept new sessions. After the transfer is complete (this process normally takes between a few seconds to a minute, depending on the storage backend, number of sessions and network throughput), Server A will notify all cluster members that it has been replaced by Server B.

From this moment and until the last session from Server A expires, all servers in the cluster will know to relay requests for sessions owned by Server A to Server B. Upon receiving such request, Server B updates the Session ID to designate that it is the new owner of the session, and the relaying and lookup process will no longer be needed by this session.

Eventually, all sessions previously owned by Server A will either expire or become owned by Server B.

Graceful Startup

But what happens if after a few minutes Server A comes back to life? This scenario is likely for example, when you need to temporarily stop a server for maintenance purposes.

In this case, when Server A comes back to life, it will check with other cluster members to see if it has been replaced during the graceful shutdown process. If so (and there are still active sessions that are marked as belonging to A), Server A will connect to Server B again, and will ask to receive all sessions owned by it back. Again, a transfer process which may take up to a minute begins, and at the end of it, Server A will notify all cluster members that it is up again – and that they can stop relaying requests to Server B.

Appendix: Tuning Session Clustering

The following appendix lists out some possibly useful tuning and configuration options for Session Clustering.

It is highly important to make sure that when modifying any of these options, configuration is modified consistently across the entire cluster. Usually, Zend Server Cluster Manager will enforce such consistency.

All the directives mentioned here are in the *etc/scd.ini* file under your installation directory. Some of these may be modified using the GUI.

Tuning Session Clustering for Performance

- Disk vs. Memory storage – set the *zend_sc.storage.use_permanent_storage* directive to 0 to use memory storage, or 1 to use disk storage. Memory storage will provide better performance, at the expense of memory consumption and storage space limitation.

  It should be noted that switching from disk to memory storage will delete all current sessions.
- Number of worker threads to launch – in most cases this will not be required, but if you notice slow session handling performance yet Session Clustering Daemons are not utilizing almost any CPU, you may want to try and increase the value of *zend_sc.number_of_threads* to a higher value.

When using disk storage, additional tuning options are available:

- *zend_sc.storage.memory_cache_size* - sets the size of memory to use for caching sessions before writing them to disk. If you have large sessions, or want to allow for more caching (at the expense of potential data loss in case of server crash), you may want to increase this value.
- *zend_sc.storage.flush_delta* – sets the number of seconds after which data is flushed from the memory cache to disk. Higher values mean less writes to disk, which may improve performance.
- *zend_sc.storage.dir_levels* – sets the directory nesting level for storing sessions on disk. For example, if set to 2, sessions with IDs starting with "abcd" will be stored in *sessions/ab/cd/abcd…* Deeper nesting levels means more *stat* system calls, but less files stored in each directory. Under most circumstances the default value (2) is fine – but if you use a file system capable of dealing with tens of thousands of files in the same directory (such as XFS in Linux), you may reduce this value. If you have an exceptionally large number of active sessions (say 50,000+), you may want to increase this.

Other Important Configuration Options

- *zend_sc.session_lifetime* – the session lifetime, in seconds. This sets the no-activity timeout after which a session becomes invalid. The default is 1440 (20 minutes), but this can be adjusted based on your application's needs.
- *zend_sc.ha.use_broadcast* – sets whether UDP broadcast should be used when sending messages to other cluster members. Normally, this should be enabled – but if your network does not allow UDP broadcast (for example, in some virtual networks or Cloud networks UDP broadcast will not function), you should set this to '0'.
- *zend_sc.ha.broadcast_address* – If using UDP broadcast, you can set your network's broadcast address here. Normally the default value (the global broadcast address, 255.255.255.255) will work, but when running multiple subnets on the same VLAN you may want to reduce this to only broadcast to a specific subnet.

[i] The memcached project: http://memcached.org/

[ii] The one potential case where code changes will be required is when the session ID is manually set or customized by the application, using the *session_id()* PHP function. This kind of modification is rare, but possible, and Session Clustering will not work well with it.

[iii] Session Clustering Daemons will keep connections between them open, so in most cases re-connection is not required. If a connection between two daemons is closed, it will be reopened at this stage.

Corporate Headquarters: Zend Technologies, Inc. 19200 Stevens Creek Blvd. Cupertino, CA 95014, USA • **Tel** +1-408-253-8800 • **Fax** +1-408-253-8801
Central Europe: (Germany, Austria, Switzerland) Zend Technologies GmbH Rosenheimer Strasse 145 b-c, 81671 Munich, Germany • **Tel** +49-89-516199-0 • **Fax** +49-89-516199-20
International: Zend Technologies Ltd. 12 Abba Hillel Street, Ramat Gan, Israel 52506 • **Tel** +972-3-753-9500 • **Fax** +972-3-613-9671
France: Zend Technologies SARL, 5 Rue de Rome, ZAC de Nanteuil, 93110 Rosny-sous-Bois, France • **Tel** +33 1 4855 0200 • **Fax** +33 1 4812 3132
Italy: Zend Technologies, Largo Richini 6, 20122 Milano, Italy • **Tel** +39 02 5821 5832 • **Fax** +39 02 5821 5400
Ireland: Zend Technologies, Regus Harcourt, Block 4, Harcourt Road, Dublin 2, Republic of Ireland • **Tel** + 353-1-4773065

© **2010 Zend Corporation.** Zend is a registered trademark of Zend Technologies Ltd.
All other trademarks are the property of their respective owners. www.zend.com

0000-0110-R2-EN