NativeSPClustering

The goal of a typical deployment is **NOT** to cluster the SP. That's a possible **solution** to solving the underlying problem, which is to cluster the application itself.

Clustering the SP requires that you understand a number of inter-related issues:

- Is the application itself clusterable?
- How does the application's session management interact with the SP's session management, if at all?
- What are the capabilities of your load balancer?
- What are your tolerances for single points of failure?

The key issue is session management. The SP does maintain other information in memory, but generally this only includes the replay cache, and the protection you get from replay checks probably isn't enough to justify worrying about clustering it.

Session management is critical because most web applications that are "high end" enough to cluster have their own session mechanism, and do not rely on the SP session. To cluster an application like this, dealing with the SP isn't enough. You'll still have to do something about the application itself, and you should start there before you even get to the SP. In other words, if you can't cluster the application without the SP, chances are you won't be able to with it.

Having addressed that, now you can turn to the SP itself. Usually, though not always, with applications that maintain their own sessions, the SP is "bolted" onto the front of the application to protect a login script, and the rest of the interactions are handled by the application or the container session. Occasionally, you might see a hybrid design where both kinds of sessions are maintained. The usual reason is to preserve the ability to do logout through the SP. Before you bother with that, note that logout is itself very complex, and incredibly difficult to pull off. Consider whether that feature alone is worth all the trouble.

Avoidance

If you can get away with minimizing the footprint of the SP session to a short-term entry point, then the easiest way to cluster is not to do it at all. Instead, rely on your load balancer to maintain session affinity (so-called stickiness) for a minute or two. The reason for this is so that the message to the SP from the IdP containing the SAML assertion can be processed and the browser redirected to the same server that received it. That redirect will normally be to the "protected" login script and the rest of the traffic should be handled by the application session, so can be to any server.

If you're using Apache you may be able to avoid even that short term stickiness requirement by using proxy session creation.

Now, let's say you can't do that. Now you have to actually cluster the SP's session store. There are a couple of ways to do this. One of them is fairly simple, but requires a private network between the cluster members, and maintains a single point of failure. The other is much more work, and is most likely also going to be a single point of failure.

Shared Process

The "simple" solution is to take advantage of the fact that the SP is divided into two pieces, and all of the session state is maintained in the `shibd` process rather than the web server. While the SP installation requires that you install both halves on each machine, you don't actually have to use both halves on each server. If you have a fast enough, and secure enough, network, you can utilize a TCP connection to connect a number of web servers running the SP to a single `shibd` "listener" process. This process can run on any of the cluster nodes, or on a separate box devoted to it. To set this up, just follow the documentation for using the TCP Listener plugin.

On Windows, this plugin is already the default, so it's just a matter of configuring it. On Unix, you'll usually have to switch to it from the "Unix" variant. Normally, the listener component binds itself to the local loopback address and blocks traffic from any other source. Just configure the machine running `shibd` and set the listener's address to an actual network address, and set the ACL to a list of addresses corresponding to your web servers. Each of them in turn has the SP installed with the same configuration, which allows them to connect to the shared process.

The overhead of this approach has not been studied extensively, but the 2.0 design was substantially changed in order to minimize the network traffic needed. For a typical application not under extreme load, this is likely to be viable. However, you have to understand a few things:

- The protocol between the servers is **NOT** secure. It's a simple XML protocol running over TCP. You **MUST** rely on a secure network between the servers, ideally a private subnet.
- The server running `shibd` is a single point of failure. If the process fails or the server fails, you'll lose all active sessions. However, you **can** restart any of the web servers without losing any state. They will reload any sessions as needed.
- Session affinity is still important here. You'll get much better performance if you keep some locality of reference, because the sessions are cached in the web servers as well.

Shared Database

If you **really** want to try to cluster the SP in a persistent way, there is a plugin provided for this purpose based on using an ODBC-compliant database. You'll see there are lots of caveats recorded there.

Keep in mind that unless you actually cluster the database itself, you still have a single point of failure. This is pretty stupid, in most cases. The only advantage you'll be getting over the shared process approach is that it should be able to recover sessions from disk if the database fails temporarily. But in practice, by the time the failure is corrected, chances are most of your users tried to restart their sessions anyway.

Clustering databases is certainly possible, but it can be very difficult and even expensive. To cluster this kind of data, you either need very rapid replication, or you'll have to get around that with session affinity.