An Efficient Approach for State Sharing in Server Pools

Thomas Dreibholz
University of Essen, Institute for Experimental Mathematics
dreibh@exp-math.uni-essen.de

Abstract

Many Internet services require high availability. Server pooling provides a high availability solution using redundant servers. If one server fails, the service is continued by another one. A challenge for server pooling is efficient state sharing: The new server requires the old one's state to continue service. This paper proposes a simple, efficient and scalable solution, usable for a large subset of applications.

1 Introduction

Server pooling (see [1, 2]) provides high availability for critical applications using redundant servers. In case of a server failure, a client can choose another server to continue operations. This procedure is called failover. A server usually requires some state for each client (e.g. media file name and media position for a video server). Therefore, it is mandatory that for a successful failover the new server has knowledge about the old server's state. A trivial solution would be to synchronize all clients' states with all servers, but it lacks of scalability.

This paper presents a simple, efficient and scalable extended cookie approach to let the client application itself do the state transfer. Cryptographic methods ensure confidentiality, integrity and authenticity of the state.

This work is part of KING, a research project of Siemens AG. The work of this project is partially funded by the Bundesministerium für Bildung und Forschung of the Federal Republic of Germany (Förderkennzeichen 01AK045).

2 Client-Based State Sharing

For better understandability, it is first necessary to describe an example scenario: an electronic shop with 3D user interface. Customers connect to a shop server using special client software allowing the user to move within the 3D shop scenario and select products into a virtual shopping cart. In case of server failure, a failover to another server is made. In this case, it is necessary to transfer the server state containing position and rotation angle of the user within the 3D scenario, quality setting for the 3D scenario (e.g. from modem connection to T1), shopping cart content including discounts granted by shop assistants to the specific customer and customer identification and accounting information (e.g. the customer's credit rating).

An observation of these sub-states leads to application-specific classifications: Some of them change very frequently (called **short-term sub-states**), e.g. position and rotation angle. Others usually remain constant (called **long-term sub-states**), e.g. the customer identification. Furthermore, states may be confidential from the client (called **private sub-states**), e.g. internal accounting information and customer's credit rating or not confidential (called **public sub-states**), e.g. quality setting, position and rotation angle. For public sub-states, a sub-grouping is possible: A mutable sub-state may be changed during failover within a well-defined range. For example, quality q_1 may be requested from the first server but at failover to another server, it may be changed to q_2. On the other side, immutable sub-states may not be changed during failover, e.g. the content of the shopping cart (containing special discounts).

Now, a server state can be formally described as $S = (S^{LT}, S^{ST})$ with long-term sub-state $S^{LT} = (S_M^{LT}, S_I^{LT}, S_P^{LT})$ and short-term sub-state $S^{ST} = (S_M^{ST}, S_I^{ST}, S_P^{ST})$, where S_M^{LT} / S_M^{ST} are the mutable, S_I^{LT} / S_I^{ST} the immutable and S_P^{LT} / S_P^{ST} the private sub-states.

For client-based state sharing, the server state S now has to be transmitted to the client on every change. Since it would be very inefficient to transmit the complete state including unchanged parts on every change, long-term and short-term part are handled separately: On every change of a long-term part, the complete state S is transmitted as so called **full state cookie**. But on changes restricted to the short-term part, only S^{ST} is transmitted as **partial state cookie**.

To avoid replay attacks and ensure correct recombination, both parts (S^{LT} and S^{ST}) require validity range v^{LT} and v^{ST} and sequence numbers σ^{LT} and σ^{ST}. The validity range gives the time range for which the state is valid (e.g.

"29-Sep-2002 10:05 to 29-Sep-2002 10:10 CET"). The sequence number differentiates sub-states having the same validity range. Using validity range and sequence number, it is always possible to reconstruct the correct order of the state cookies. The partial state cookie further requires a reference ρ^{LT} to the long-term part it belongs to. This could simply be the tuple $\rho^{LT} = (\nu^{LT}, \sigma^{LT})$.

Now, it is possible to transmit the state S to the client and keep it up-to-date there. On failover, it can be transmitted to the new server and restored. But confidentiality for private sub-states and a check for integrity and authenticity at the new server are still missing. These functionalities can be realized easily using a digital signature and encryption. That is, all servers of a pool need the same pool key K, e.g. a random value that is changed every 3 hours to increase security. Assuming that a full state cookie is sent on every key change, only the full state cookie has to store a key reference λ. That is a key ID that states which key to use (still the old or already the new one during a key change phase). All immutable sub-states, validity ranges, sequence numbers and references can now be digitally signed using a signature function φ_λ to ensure their authenticity and integrity. An encryption function ψ_λ additionally ensures confidentiality of the private sub-states. Note, that mutable sub-states are not signed or encrypted, since changes within well-defined ranges are allowed.

Client-based state sharing is usable for all applications that can cope with the following two limitations: First, it allows reverting to an older state within its validity range. That is, server A changes from state S_1 to state S_2, but the client sends S_1 to the new server B during failover. Then, B restores the older state S_1 as long as its validity range is still accepted. This prevents applications like money transfer from using client-based state sharing but most applications like media streaming (e.g. older media position) or the example 3D shop (older position, rotation angle or shopping cart content) are uncritical. The second restriction is possible forking. That is, the client can send a valid state to more than one server and create multiple new sessions. Note, that a malicious user can be prevented easily from forking a session using a sniffed state by requiring the client to authenticate to the server. Therefore, forking is usually not critical, too. Since these restrictions are acceptable in most cases, scalable, simple and efficient client-based state sharing is applicable for a large subset of server pooling applications.

3 Optimizations

Some optimizations to client-based state sharing are possible to reduce bandwidth requirements and state transmission effort, depending on application requirements:

State Splitting: Analogous to the separation of sub-states into long-term and short-term ones, a finer classification depending on their change frequency is possible. This leads to a sub-state hierarchy having the most frequently changed sub-states in the lowest and the rarely changed ones in the highest group. Anytime a state in an upper group changes, all lower groups also have to be transmitted to ensure correct references for recombination.

State Derivation: The client may be able to derive public mutable states directly from the user data (e.g. a media position from the RTP timestamp) instead of requiring its separate transmission. In the public/mutable-only case with all states derivable, no separate server to client transmission of states is necessary at all!

State Approximation: Depending on the application, an approximation of states may be sufficient. For example for a video server using 25 frames per second, the media position changes 25 times per second and requires the same number of transmissions of the short-term sub-state. Reducing this to once per second would only lead to a repetition of at most one second in the rare case of a server failure (e.g. once per week or month). Since the failover is visible to the user anyway (e.g. as a short picture freeze), this should be no problem.

4 Summary and Conclusions

This paper has presented an efficient, simple and scalable approach for state sharing in server pools, applicable to a large subset of applications. The server state is transmitted to the client using an extended cookie mechanism and is kept up-to-date there. On failover, the client itself can provide the state to the new server. Integrity, authenticity and confidentiality of the state are ensured using digital signature and encryption. Finally, optimizations to reduce bandwidth and state transmission effort have been presented. Further work will include the formal definition of a state sharing protocol and finally its standardization by the IETF.

References

[1] RSerPool Introduction
`http://tdrwww.exp-math.uni-essen.de/dreibholz/rserpool/index.html`

[2] Architecture for Reliable Server Pooling
draft-ietf-rserpool-arch-03.txt