Network Working Group
Internet-Draft
Expires: April 9, 2006

P. Lei
Cisco Systems
P. Conrad
University of Delaware

October 6, 2005

Services Provided By Reliable Server Pooling
draft-ietf-rserpool-service-02.txt

Status of this Memo

Copyright Notice

Abstract

The Reliable Server Pooling architecture (abbreviated "RSerPool", and defined in [3]), provides a set of services and protocols for building fault tolerant and highly available client/server applications. This memo describes the semantics of the services that RSerPool provides to upper layer protocols.

Table of Contents

1. Introduction

The Reliable Server Pooling architecture is defined in [3]. The central idea of this architecture is to provide client applications ("pool users") with the ability to select a server (a "pool element") from among a group of servers providing equivalent service (a "pool"). The pool is accessed via an identifier called a "pool handle". The RSerPool architecture supports high-availabilty and load balancing by enabling a pool user to identify the most appropriate server from the server pool at a given time. The architecture also supports failover to an alternate server when needed.

This memo describes how an upper layer protocol or application for a pool user or pool element uses the RSerPool architecture and protocols. Specifically, it describes how the ASAP protocol [5] and transport protocols (SCTP, TCP, etc.) can be utilized to realize highly available services between pool users and pool elements.

The purpose of this document is to describe:

1. the precise services provided by RSerPool to the upper layer,

2. the tradeoffs in choosing which services to utilize,

3. how applications must be designed for each of these services,

4. how applications written over various transports (SCTP, TCP, and others) can be mapped into these services.

RSerPool services can be used in one of two modes: "Basic Mode" and "Enhanced Mode". Basic Mode provides a smaller set of services than Enhanced Mode, but offers imposes fewer restrictions on the application layer protocols that can be supported. Enhanced Mode provides extra capabilities, including some features that require applications to exchange application data messages via RSerPool service primitives (a restriction not present in Basic Mode).

For Enhanced Mode, the RSerPool data exchange primitives are implemented by multiplexing the ASAP messages and application data over a single transport protocol connection or association. This memo defines how to do this multiplexing over SCTP. This memo also describes the requirments needed to extend support to other transport protocols as required.

Note that while RSerPool services are divided into Basic and Enhanced Modes, both modes assume a full implementation of the ASAP protocol. The purpose of dividing RSerPool services into two modes is solely to

provide more flexibility for applications to interact with RSerPool. In particular, Basic Mode provides an easy migration path for legacy applications to take advantage of many useful RSerPool services, including load balancing and high availability. Enhanced Mode extends the services provided by Basic Mode with enhanced failover capabilities.

2. Conventions Used In This Document

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [2].

3. Example Application Scenarios

To illustrate the differences between Basic and Enhanced Mode, this section decribes example failover scenarios for:

an application written directly over a transport layer protocol, not utilizing RSerPool

an application using RSerPool Basic Mode, and

an application using RSerPool Enhanced Mode.

3.1. Example Scenario for Failover Without RSerPool

Consider a typical client/server application that does not use a reliable server pooling framework of any kind. Typically, the server is specified by a DNS name. At some point, the application translates this name to an IP address (via DNS), and subsequently makes initial contact with the server to begin a session, via SCTP, TCP, UDP, or other transport protocol. If the client loses contact or fails to make contact with the server (either due to server failure, or a failure in the network) the client must either abandon the session, or try to contact another server.

In this scenario, the client must first determine that a failure took place. There are several ways that a client application may determine that a server failed, including the following:

1. The client may have sent a request to the server, and may time out waiting for a response, or may receive a message such as "no route to host", "port not available", or "connection refused".

2. The client may have sent a request to the server, or may have tried to initiate a connection or association and may have received a connection/association failure error.

3. The client may already have established a connection to server, but at some point receives an indication from the transport layer that the connection failed.

Suppose that the client application has a feature by which the user can enter the hostname of a secondary server to contact in the event of failure. Once the application determines that a failure took place on the primary server, the application can then attempt to resolve the hostname of the secondary server, and contact the secondary server to establish a session there. This process can be iterated to a tertiary server, and so forth.

In this scenario, the identification of these alternate servers is an additional burden placedd on the end user. Furthermore, there is no capability in this model to dynamically update the identity of the alternate servers based on current availablity or reachability. DNS has some capabilities that can be used to help, but there are significant limitations to these capabilities (See [4] for a discussion of this point).

3.2. Example Scenario Using RSerPool Basic Mode

Now consider the same client/server application mentioned in Section 3.1. First we describe what the application programmer must do to modify the code to use RSerPool Basic Mode. We then describe the benefits that these modifications provide.

For pool user ("client") applications, if an ASAP implementation is available on the client system, there are typically only three modifications required to the application source code:

1. Instead of specifying the hostnames of primary, secondary, tertiary servers, etc., the application user specifies a pool handle.

2. Instead of using a DNS based service (e.g. the Unix library function gethostbyname()) to translate from a hostname to an IP address, the application will invoke an RSerPool service primitive GETPRIMARYSERVER that takes as input a pool handle, and returns the IP address of the primary server. The application then uses that IP address just as it would have used the IP address returned by the DNS in the previous scenario.

3. Without the use of additional RSerPool services, failure detection is application specific just as in the previous scenario. However, when failure is detected on the primary server, instead of invoking DNS translation again on the hostname of a secondary server, the application invokes the service primitive GETNEXTSERVER, which performs two functions in a single operation.

    1. First it indicates to the RSerPool layer the failure of the server returned by a previous GETPRIMARYSERVER or GETNEXTSERVER call.

    2. Second, it provides the IP address of the next server that should be contacted, according to the best information available to the RSerPool layer at the present time (e.g. set of available pool elements, pool element policy in effect for the pool, etc.).

For pool element ("server") applications where an ASAP implementation is available, two changes are required to the application source code:

1. The server should invoke the REGISTER service primitive upon startup to add itself into the server pool using an appropriate pool handle. This also includes the address(es) protocol or mapping id, port (if required by the mapping), and pooling policy(s).

2. The server should invoke the DEREGISTER service primitive to remove itself from the server pool when shutting down.

When using these RSerPool services, RSerPool provides benefits that are limited (as compared to utilizing all services, described in Section 3.3), but nevertheless quite useful as compared to not using RSerPool at all (as in Section 3.1). First, the client user need only supply a single string, i.e. the pool handle, rather than a list of servers. Second, the decision as to which server is to be used can be determined dynamically by the server selection mechanism (i.e. a "pool policy" performed by ASAP; see [3]). Finally, when failures occur, these are reported to the pool via signaling present in ASAP [5]) and ENRP [6], other clients will eventually know (once this failure is confirmed by other elements of the RSerPool architecture) that this server has failed.

Utilizing this subset of services is useful for:

applications built over connectionless protocols such as UDP that cannot easily be adapted to the transport layer requirements

required for enhanced services (see section Section 5)

applications running on systems which do not provide an appropriate mapping layer for the desired transport protcol

an expedient way to provide some of the benefits of RSerPool to legacy applications (regardless of the transport protocol used)

However, to take full advantage of the RSerPool framework, utilization of the complete set of Enhanced Mode services as described in the next section is recommended.

3.3. Example Scenario Using RSerPool Enhanced Mode

Finally, consider the same client/server application as in Section 3.1, but this time, modified to take advantage of RSerPool Enhanced Mode services. As in the Section 3.1, we first describe the modifications needed, then we describe the benefits provided.

When the full suite of RSerPool services are used, all communication between the pool user and the pool element is mediated by the RSerPool framework, including session establishment and teardown, and the sending and receiving of data. Accordingly, it is necessary to modify the application to use the service primitives (i.e. the API) provided by RSerPool, rather than the transport layer primitives provided by TCP, SCTP, or whatever transport protocol is being used.

As in the previous case, sessions (rather than connections or associations) are established, and the destination endpoint is specified as a pool handle rather than as a list of IP addresses with a port number. However, failover from one pool element to another is fully automatic, and can be transparent to the application:

The RSerPool framework control channel provides maintainance functions to keep pool element lists, policies, etc. current.

Since the application data (e.g. data channel) is managed by the RSerPool framework, unsent data (data not yet submitted by RSerPool to the underlying transport protocol) is automatically redirected to the newly selected pool element upon failover. If the underlying transport layer supports retrieval of unsent data (as in SCTP), retrieved unsent data can also be automatically re-sent to the newly selected pool element.

An application server (pool element) can provide a state cookie (described in Section 4.5.1) that is automatically passed on to another pool element (by the ASAP layer at the pool user) in the event of a failover. This state cookie can be used to assist the

application at the new pool element in recreating whatever state is needed to continue a session or transaction that was interrupted by a failure in the communication between a pool user and the original pool element.

The application client (pool user) can provide a callback function (described in Section 4.5.2) that is invoked on the pool user side in the case of a failover. This callback function can execute any application specific failover code, such as generating a special message (or sequence of messages) that helps the new pool element construct any state needed to continue an in-process session.

Suppose in a particular peer-to-peer application, PU A is communicating with PE B, and it so happens that PU A is also a PE in pool X. PU A can pass a "business card" to PE B identifying it as a member of pool X. In the event of a failure at A, or a failure in the communication link between A and B, PE B can use the information in the business card to contact an equivalent PE to PU A from pool X.

Additionally, if the application at PU A is aware of some particular PEs of pool X that would be preferred for B to contact in the event that A becomes unreachable from B, PU A can provide that list to the ASAP layer, and it will be included in A's business card. (See Section 4.5.3)).

Retrofitting an existing application for Enhanced Mode requires more application programmer effort than retrofitting an application for Basic Mode. In particular, all use of the transport layer's primitives (e.g. the calls to the sockets API) must be replaced by the use of the RSerPool primitives (e.g. the RSerPool API). This can be mitigated by making the RSerPool API as close to existing transport APIs as possible. However, the benefit is that failure detection and failover is automated in this case. This automatic failure detection takes advantage of heartbeat mechanisms that are provided either in the underlying transport protocol, or in a mapping defined on top of that protocol (see Section 4.5).

Provided that developers of APIs for RSerPool stay close to familiar APIs for existing transport protocols, the effort of writing a new applications over RSerPool Enhanced Mode need not be significantly different from writing the same application directly over a supported transport protocol or mapping.

4. Service Primitives

Upper layer protocols and applications may "choose" to use these

primitive services as needed. By selecting and using the appropriate set of service primitives, a range of failover scenarios may be supported. These service primitives are described in the sub-sections that follow.

4.1. Initialization

The INITIALIZE service is used to establish a service access point to communicate with the ASAP layer on the local host. This is the first service accessed by either a PU or a PE.

4.2. PE Registration Services

Pool Elements ("server") must use the following services to add or remove themselves from server pools:

REGISTER, to add the pool element into a server pool using {pool handle, mapping mode, protocol or mapping id, port, policy info} where mapping mode is defined in Section 5. A response result code is returned.

DEREGISTER, to remove the pool element from a server pool using {pool handle, mapping mode, protocol or mapping id, port, policy info} where mapping mode is defined in Section 5. A response result code is returned.

4.3. PE Selection Services

When automatic failover is enabled, selection of a new pool element according to the pool policy in place is automatically performed by the RSerPool framework in case of a detected failure (e.g. provides automatic failover). No application intervention is required.

Automatic failover may be enabled by setting the appropriate send flag when used in conjuction with data channel services (described in Section 4.4) or explicitly during initialization when data channel services are not used.

FAILOVER_INDICATION, delivered by callback, indicates that a failover has occurred and that any required application level state recovery should be performed. The newly selected pool element handle is provided.

Business Card services: when automatic failover is used, the exchange of business cards for rendezvous services is automatically performed by the RSerPool framework (e.g. no application intervention is required.

When automatic failover is not enabled, failover detection and selection of an alternate PE must be done by the upper layer/ application. The following primitives are provided:

GET_PRIMARY_SERVER, takes as input a pool handle and returns the {IP address, transport protocol, transport protocol port} of the primary server.

GET_NEXT_SERVER has a dual meaning. First, it indicates to the RSerPool layer the failure of the server returned by a previous GET_PRIMARY_SERVER or GET_NEXT_SERVER call. Second, it provides the {IP address, transport protocol, transport protocol port} of the next server that should be contacted, according to the best information available to the RSerPool layer at the present time. The appropriate pool policy for server selection for the pool should be used for selecting the next server.

4.4. RSerPool Managed Data Channel

The RSerPool framework provides these services to send and receive application layer data, which are used in place of the direct call of transport level system functions (e.g. send/sendto, recv/recvfrom) and provides additional functionality to those calls.

DATA_SEND_REQUEST, to send data to a pool element by using a pool handle, specific pool element handle, or by transport address. When sending to a pool handle, the specific pool element handle chosen is returned. In the case that data is sent to a pool handle, or specific pool element handle, the user can request automatic resending (on a best-effort basis) if the original pool element selected is unreachable. (However, it is ultimately the application's responsibility to detect and recover from errors, using acknowledgements at the application layer if needed.)

When sending to a specific transport address, this primitive is considered a "pass thru" to the underlying transport, and no failover services are performed.

In each case, appropriate error code(s) are returned in the event of failure. (see [5] for more detail).

DATA_RECEIVED_NOTIFICATION, delivered by callback, to indicate that data has been received from a pool element and to pass that data to the application layer protocol. An application layer acknowledgement request can be indicated along with the data.

4.5. Failover Services

The charter of the RSerPool Working Group specifically states that transaction failover is out of scope for RSerPool, i.e. "if a server fails during processing of a transaction this transaction may be lost. Some services may provide a way to handle the failure, but this is not guaranteed." Accordingly, the RSerPool framework provides three "hooks" for applications to provide their own application-specific failover mechanism(s), one on the PE side (State Cookie Exchange), one on the PU side (Failover Callback), and one for entities that are combination of PU/PE (business card).

4.5.1. State Cookie Exchange

SET_COOKIE: This is invoked by a PE to set the state cookie that is sent periodically over the control channel, when present, from a PE to a PU. The most recently received cookie is cached by the PU; in the event of failover, it is forwarded to the new PE.

COOKIE_INDICATION: This is invoked by callback at a PE, when that PE receives a cookie from a PU. This cookie is an indication that the PU has failed over to the current PE from some other PE. The contents of the cookie are provided to the PE prior to any data.indication for messages arriving from the PU that sent the cookie. This provides a hook by which a PE "X" can send a "hint" to its successor PE "Y", in the event that one of X's PU's fails over from X to Y. PE "Y" can use the contents of the cookie to establish application layer state prior to processing resent or new messages from the PU. The PU application layer is not involved in any way in this exchange; it is handled automatically by the ASAP layer.

4.5.2. Failover Callback Function

AUTHOR'S NOTE (PTC): the service defined in this section is not a part of section 4 of the current version ASAP draft. It should either be added to the ASAP draft, or this service should be removed from the services draft, after discussion on the list. Open question: does the cookie feature eliminate the need for this feature?

An PU that establishing a session with a PE can specify a callback function that is invoked whenever a failover has taken place. This callback function is invoked immediately after the new transport layer connection/ association is established with a new server, and gives the application the opportunity to send one or more messages that may help the server to resume any transaction or session that was in progress when the first server failed. In essence, this

allows an application designed to put the reestablishment of state into the PU side instead of the PE side, if desired.

This service that complements the cookie feature, in the following way: the cookie feature provides failover hooks on the PE side, where the callback is a failover hook for the PU side. The on-the-wire impact is that it is important that the ASAP entity should invoke the failover callback (if any is registered) prior to resending any messages from previous DATA_SEND_REQUEST primitives.

Note that if both a state cookie from a PU and a failover callback are present, the state cookie should be sent before the failover callback is issued.

As a simple example of how such a callback is useful, consider a file transfer service built using RSerPool. Let us assume that some FTP mirroring software is used to maintain mirrored sites, and that the actual mirroring is out of scope. However, we would like to use RSerPool to select a server from among the available mirror sites, and to failover in the middle of a file transfer if a primary server fails.

For this example, assume that a simple request/response protocol is used, where one request message results in one or more response messages. Each request message contains the filename, and the offset desired within the file, (default zero.) Each response message contains some portion of the file, along with the offset, length of the portion in this message, and the length of the entire file.

A single request is sufficient to result in a sequence of response messages from the requested offset to the end of the file.

In this protocol, all that is needed for failover is for the application to:

- keep track of the lowest byte that it has not yet received from the server,

- provide a callback function that reissues the request to the new server, replacing the offset with this number.

When there is no failover, only one request message is sent and the minimum number of response messages are returned; in the event of failover(s), single new request message is sent for each failover that occurs.

While this is a simple example, for more complex application requirements, the failover callback could be used in a variety of

ways:

The client might send security credentials for authentication by the server, and/or to provide a "key" by which the server could locate and setup state by accessing some application-specific (and out-of-scope) state sharing mechanism used by the servers.

The client might keep track of various synchronization points in the transaction, and use the failover callback to replay message from a recent synchronization point.

4.5.3. Business Card

This section TBD... describe a service primitive Set.BusinessCard.PE.List. What should the parameters be? This service primitive should also be defined in Section 4 of the ASAP draft.

5. Transport Mappings

While SCTP is the preferred transport layer protocol for applications built for RSerPool failover mode (for reasons explained shortly), it is also possible to use other transport protocols as well (e.g. TCP) if an SCTP implementation is not available on the client and/or server. However, there are certain features present in SCTP that are required if the RSerPool framework is to function in failover mode. When a transport protocol other than SCTP is used, these features must be provided by an "adaption layer" (also called a "shim protocol") that sits between the base transport protocol (e.g. TCP) and the RSerPool layer. We refer to these "adaptation layers" or "shim protocols" as "mappings" as the idea is that the requirements of the RSerPool framework are "mapped" onto the capabilities of the underlying protocol (e.g. SCTP or TCP).

5.1. Defined Transport Mappings

In order to support the RSerPool framework over a variety of transport protocols and configurations, several mappings are defined to provide RSerPool services over a given transport protocol. Each mapping translates the requirements of the RSerPool framework onto the capabilities of the transport protocol desired (e.g. SCTP, TCP, etc.). Initially, three mappings are defined:

NO_MAPPING (0x00): With this mapping, no RserPool control channel is provided and the application specific communication between a pool user and the pool element (e.g. data channel) is out of scope of RSerPool. However, pool elements can register the application

specific communication "protocol" and "port", and thus can be provided to pool users.

SCTP (0x01): SCTP transport is used for the RSerPool control channel. The data channel MAY be multiplexed onto the same SCTP association, if desired. This mapping is the preferred mapping.

TCP (0x02): TCP transport is used for the RSerPool control channel. The data channel MAY be multiplexed onto the same TCP connection, if desired.

A particular pool element might support any combination of these mappings in order to support a variety of pool users with different capabilities (i.e. different mapping support). In this case, pool elements should register each mapping that it supports with its pool(s).

5.2. Transport Mappings Requirements

5.2.1. Mappings: Mandatory Requirements

These features MUST be present in any mapping of the RSerPool framework mode to TCP (or any other transport protocol):

1. Message orientation, which facilitates application re-synchronization during failover. Messages must be "framed" in order to allow for undelivered message retrieval from the transport protocol.

2. A heartbeat mechanism to monitor the health of an association or connection.

3. A mechanism to transport and differentiate between control channel messages (e.g. ASAP messages) and data channel messages. For example in SCTP, the payload protocol identifier (PPID) may be used.

4. [NOTE: retrieval was eliminated here as a requirement, now that failover is best effort.]

5.2.2. Mappings: Optional Requirements

There are several additional features that are present in SCTP that are lacking in TCP. While these features are not crucial to RSerPool, providing them in the mapping layer makes it easier for an application layer programmer to write to a single API. This single API can then be mapped over both SCTP and TCP, as well as any other transport protocol for which a mapping is provided. Since these

features are not essential for RSerPool, they are optional in any defined mapping. However, appropriate error messages or indications should be provided when these features are not available. These features include:

1. Support for multiple streams

2. Support for unordered delivery of messages

5.2.3. Mappings: Other Requirements

There are some features of SCTP that a mapping may not be able to provide, because they would require access to transport layer internals, or modifications in the transport layer itself. The services provided by the RSerPool layer to the application should therefore provide mechanisms for the upper layer to access these features when present (e.g. in SCTP), but also provide appropriate error messages or indications that these features are not available when they cannot be provided. These features include:

1. Application access to the RTT and RTO estimates

2. Application access to the Path MTU value

3. Application access to set the lifetime parameter on outgoing SCTP messages

6. Security Considerations

[Open Issue TBD: Security issues are not discussed in this memo at this time, but will be added in a later version of this draft.]

7. IANA Considerations

[Open Issue TBD: Will there be an enumeration of the various transport layer mappings that must be registered with IANA?]

8. Acknowledgements

The authors wish to thank Maureen Stillman, Qiaobing Xie, Michael Tuexen, Randall Stewart, and many others for their invaluable comments.

9. References

[1] Bradner, S., "The Internet Standards Process -- Revision 3", BCP 9, RFC 2026, October 1996.

[2] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[3] Tuexen, M., "Architecture for Reliable Server Pooling", draft-ietf-rserpool-arch-10 (work in progress), July 2005.

[4] Loughney, J., "Comparison of Protocols for Reliable Server Pooling", draft-ietf-rserpool-comp-10 (work in progress), July 2005.

[5] Stewart, R., "Aggregate Server Access Protocol (ASAP)", draft-ietf-rserpool-asap-12 (work in progress), July 2005.

[6] Stewart, R., "Endpoint Handlespace Redundancy Protocol (ENRP)", draft-ietf-rserpool-enrp-12 (work in progress), July 2005.

Authors' Addresses

Peter Lei
Cisco Systems
8735 W Higgins Rd, Suite 300
Chicago, IL 60631
US

Phone: +1 773 695 8201
Email: peterlei@cisco.com

Phillip T. Conrad
University of Delaware
Dept. of Computer and Information Sciences
103 Smith Hall
Newark, DE 19716
US

Phone: +1 302 831 8622
Email: conrad@acm.org
URI: http://udel.edu/~pconrad

Intellectual Property Statement

Disclaimer of Validity

Copyright Statement

Acknowledgment

Funding for the RFC Editor function is currently provided by the Internet Society.