Network Working Group
Internet-Draft
Expires: May 2, 2003

P. Conrad
Temple University
P. Lei
Cisco Systems, Inc.

November 1, 2002

Services Provided By Reliable Server Pooling
draft-conrad-rserpool-service-03.txt

Status of this Memo

Copyright Notice

Abstract

RSerPool [1] is a framework to provide highly available services between clients and servers. This is achieved by grouping servers into pools, each with an identifier and pooling policy. Three classes of entities are defined: Pool Users (clients), Pool Elements (servers), and Name Servers.

This memo defines the services provided by this framework to upper layer protocols and applications for Pool Users and Pool Elements. It describes the service primitives that the framework provides and describes example scenarios.

It also describes the requirements for mapping (or adaption or "shim") layers for a variety of transport protocols (SCTP, TCP, or others) such that upper layer protocols and applications may use a common framework/API to utilize the services provided.

Table of Contents

1.   Introduction 3
2.   Conventions Used In This Document 4
3.   Example Application Scenarios 4
3.1   Example Scenario for Failover Without RSerPool 4
3.2   Example Scenario Using RSerPool Name Services Only 5
3.3   Example Scenario Using Full RSerPool Services 7
4.   Service Primitives 8
4.1   Initialization 8
4.2   PE Registration Services 8
4.3   Failover Callback Function 9
4.4   PE Selection Services 10
4.5   Upper Layer/Application Level Acknowledgements 11
4.6   RSerPool Managed Data Channel 11
5.   Transport Mappings 12
5.1   Defined Transport Mappings 12
5.2   Transport Mappings Requirements 13
5.2.1 Mappings: Mandatory Requirements 13
5.2.2 Mappings: Optional Requirements 13
5.2.3 Mappings: Other Requirements 14
6.   Security Considerations 14
7.   IANA Considerations 14
8.   Acknowledgements 14
       References 14
       Authors' Addresses 15
       Full Copyright Statement 16

1. Introduction

The Reliable Server Pooling architecture is defined in [1]. The architecture provides highly available services by defining three classes of entities: pool users (clients), pool elements (servers), and name servers. Pool elements are grouped into server pools and can be used by pool users via its pool name (or "handle") and can be selected by following the pool's pool element selection policy.

This memo describes how an upper layer protocol or application for a pool user or pool element uses this architecture and associated protocols to achieve these goals described in that document. 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.

There are tradeoffs between the amount of application modification required, the features and restrictions that the underlying transport is required to support, and the richness of the feature set provided by RSerPool. In order to provide support for both existing/legacy (non-RSerPool) and new applications, several service primitives are defined in which an upper layer protocol can interact with the RSerPool framework. Depending on the number of services utilized, the upper layer protocol achieve a range of reliability from simple pool element selection to a fully automatic failover capability.

Utilizing a limited set of RSerPool services provides the capability for legacy upper layer software to use a few RSerPool services with relatively minor modifications, and allows a broad range of underlying transport protocols to be supported. To achieve a richer and more complete failover model, however, a majority of the RSerPool services should be used, which places certain requirements and restrictions on the transport layers that can be supported.

Note that regardless of the number of service primitives actually utilized by any given upper layer protocol, this document assumes that the upper layer protocol/application is operating on a platform that has a full running, implmentation of ASAP.

The following figure illustrates the protocol stacks when using the RSerPool framework (Pool Element perspective shown). Note that the mapping layer MAY be a "NULL" layer, if no control channel is utilized and/or the data channel is not utilized (e.g. application specific data).

```
+---------------------------------------+
|  Application/Upper Layer Protocol     |
+---------------------------------------+
|             RSerPool API              |
|(control channel) |  (data channel)    |
+------------------+--------+-----------+
|          ASAP layer       |           |
+------------------+--------+           |
|         mapping/adaption layer        |
+------------------+--------------------+
|           transport protocol          |
+------------------+--------------------+
```

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.

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 among an application without RSerPool, an application using limited RserPool services, and an application using a full suite of RSerPool services, this section provides an informal description of how failover may be handled in each of these cases.

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 some other 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.

A limitation of this model is that there is no provision (other than static client configuration, plus the capabilities of DNS) to determine which server to contact initially (other than DNS) or which server to contact next in the event of server failure. (See [3] for a discussion of the limitations of using DNS for this purpose.)

3.2 Example Scenario Using RSerPool Name Services Only

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 name services. We then describe the benefits that these modifications provide.

For pool user ("client") applications, there are typically only three modifications are required along with adding ASAP:

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

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 has a dual meaning. First it indicates to the RSerPool layer the failure of the server returned by a previous "GetPrimaryServer" or "GetNextServer" call. 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, two additions in are required along with adding ASAP:

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 [1]). Finally, when failures occur, these are reported to the pool via signaling present in ASAP [5]) and ENRP [4], 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 full

failover services (see section Section 5) or for 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 full suite of services as described in the next section is recommended.

3.3 Example Scenario Using Full RSerPool Services

Finally, consider the same client/server application as in Section 3.1, but this time, modified to all RSerPool provided 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 not only session establishment and teardown, but also 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, any unsent and unacknowledged transport data can be automatically re-sent to the newly selected pool element upon failover. This is enhanced by providing the application an "upper layer acknowledegment" service.

The application can provide a callback function (described in Section 4.3) that is invoked 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.

Retrofitting an existing application to this mode of RSerPool requires more effort on the part of the application programmer than retrofitting an application to use just the pool selection services; all use of the transport layer's primitives (e.g. the calls to the

sockets API) must be modified to use the RSerPool primitives (e.g. the RSerPool API). This can be mitigated by making the API for RSerPool as close to existing transport APIs as possible. However, failure detection and failover is automated in this case.

Furthermore, since the primitives provided by RSerPool are similar to those of existing transport protocols (and, it is hoped, the APIs will be also) for developers of new applications, writing to the RSerPool failover mode primitives is not significantly different in terms of programmer effort or learning curve than writing the same applications over existing transport layer primitives.

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

[OPEN TBD: what primitive(s) does a PU indicate what mappings can be used (are supported), whether automatic rollover, message retrieval are desired, etc. These will likely be in the form of a initialization call]

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.

TBD: if REGISTER also returns an opaque instance id, the application can just use that id for DEREGISTER, instead of passing in the (same) parameters used in REGISTER.

4.3 Failover Callback Function

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 a "hook" for applications to provide their own application-specific failover mechanism(s).

Specifically, an application 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.

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 results is sufficient to result in a sequence of response messages from the requested offset to the end of the file. For simplicity, assume that the response messages are delivered by the underlying transport strictly in order (although this requirement could be relaxed if a small amount of extra complexity were introduced.)

In this protocol, all that is needed for failover is for the application to keep track of the number of bytes that it has read from the server, and to 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.

[Open Issue TBD: Are there others to add to this list?]

4.4 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 4.6) 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.

4.5 Upper Layer/Application Level Acknowledgements

The RSerPool framework provides an upper layer/application level ack service. The upper layer protocol may request that the peer acknowledge receipt and successful processing of its sent data, providing an additional degree of confidence over transport level message retrieval. When used in conjuction with the data channel services (described in Section 4.6), any unacknowledged data will be automatically sent to a new pool element in case of failover, if desired (e.g. automatic failover is enabled). The following service primitive is used to acknowledge an upper layer acknowledgement request.

ULP_ACK, responds to a received upper layer acknowledgement request.

4.6 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, to send data to a pool element by using a pool handle, specific pool element handle, or by transport address. An upper layer acknowledgement may be requested with this service. Appropriate error code(s) are returned. When sending to a pool handle, the specific pool element handle is returned.

DATA_INDICATION, 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.

The application MAY direct that the RSerPool framework multiplex both the control and data channels onto the same SCTP association/TCP connection/ etc., if desired.

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 retrieval mechanism to allow an application to retrieve unsent or unacknowledged data from the transport layer upon failover.

4. 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.

5. [Open issue TBD: Are there others to be included here?]

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

3. [Open issue TBD: Are there others to be included here?]

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. [Open issue TBD: Are there others to be included here?]

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.

References

[1] Ong, L., Shore, M., Stillman, M., Xie, Q., Loughney, J., Tuexen, M. and M. Stewart, "Architecture for Reliable Server Pooling", draft-ietf-rserpool-arch-03 (work in progress), July 2002.

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

[3] Loughney, J., "Comparison of Protocols for Reliable Server Pooling", draft-ietf-rserpool-comp-04 (work in progress), July 2002.

[4] Stillman, M., Xie, Q. and R. Stewart, "Enpoint Name Resolution

Protocol (ENRP)", draft-ietf-rserpool-enrp-04 (work in progress), September 2002.

[5] Stillman, M., Xie, Q., Tuexen, M. and R. Stewart, "Aggregate Server Access Protocol (ASAP)", draft-ietf-rserpool-asap-04 (work in progress), July 2002.

Authors' Addresses

Phillip T. Conrad
Temple University
CIS Department
Room 303, Computer Building (038-24)
1805 N. Broad St.
Philadelphia, PA 19122
US

Phone: +1 215 204 7910
EMail: conrad@acm.org
URI: http://www.cis.temple.edu/~conrad

Peter Lei
Cisco Systems, Inc.
955 Happfield Dr.
Arlington Heights, IL 60004
US

Phone: +1 847 870 7201
EMail: peterlei@cisco.com

Acknowledgement

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