N/A A. Deason
Internet-Draft
SNA
Intended status: Informational April 20, 2012
Expires: October 22, 2012

**Data Transmission Over Out-of-Band Alternative Tranports for AFS-3**
**draft-deason-afs3-oob-00**

Abstract

This document describes an extension to AFS-3 which allows data transfer over arbitrary alternative transports to the traditional Rx RPC over UDP, and describes a method of using TCP as one such alternative transport. This document describes new wire structures and Rx RPCs for the 'afsint' protocol in AFS-3.

Internet Draft Comments

Comments regarding this draft are solicited. Please include the AFS-3 protocol standardization mailing list (afs3-standardization@openafs.org) as a recipient of any comments.

Status of this Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on October 22, 2012.

publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document.

Table of Contents

## 1. Introduction

AFS-3 is a global distributed network file system. Currently, most transfers between clients and servers in AFS-3 occur over an RPC-like protocol called Rx [RX] [AFS3-RX], which operates on top of UDP. For transferring bulk file data, usually the specific RPCs FetchData64 and StoreData64 are used, which allow a client to receive or send data using a stream over Rx.

Rx currently suffers from a number of performance limitations. Some of these are due to the Rx protocol design, some are due to implementation limitations in the only known implementations, and some exist simply because modern networking equipment often provide hardware acceleration or other benefits for TCP or other protocols, but provide none for UDP or Rx. Due to the widespread adoption of TCP, the comparatively less common use of UDP, and the common lack of awareness that Rx even exists, it seems unlikely that performance of Rx over UDP will ever approach that of raw TCP, except in a small amount of environments.

To improve this situation as well as other problems with the RxUDP protocol, a new protocol implementing Rx over TCP could be designed. However, designing such a protocol that covers the entire Rx protocol is a very complex task, and there are numerous obstacles that must be overcome. Some of these are caused by the difference in resources consumed by an RxUDP connection (lightweight) vs a TCP connection (relatively heavyweight), and some are simply due to the complexity of the Rx protocol, and RPC-like protocols in general.

Since high-throughput client<->server transfers in AFS-3 generally only occur via two RPCs (FetchData64 and StoreData64), we can just improve throughput for those RPCs, and the complexity of designing Rx itself to work on top of TCP or other transports can be avoided. This document describes the use of two new RxUDP RPCs called FetchDataOOB and StoreDataOOB, which act as replacements for the existing FetchData64 and StoreData64 RPCs, and transfer the relevant file data outside of Rx. This can be thought of as analagous to the "control connection" and "data conection" used by FTP [RFC0959].

## 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 [RFC2119].

## 3. Extant RPC Interface

This document references the existing RPCs FetchData64 and StoreData64. The definition for the FetchData and StoreData RPCs can be found in [AFS3-FSCM]. FetchData64 and StoreData64 behave identically to FetchData and StoreData, except that some of the arguments are expanded to be 64-bits wide instead of just 32-bits.

## 4. RPC Interface

### 4.1. FetchDataOOB

FetchDataOOB behaves very similarly to the extant RPC FetchData64. The RPC-L definition is:

```
FetchDataOOB(IN AFSFid *Fid, afs_uint64 Pos, afs_uint64 Length,
             OUT AFSFetchStatus *Status, AFSCallBack *CallBack,
             AFSVolSync *Sync) split = XXX;
```

All of the arguments are identical to those of FetchData64. This is a split call, just like FetchData64, but the data in the split call stream is not the requested file data. Instead, the contents of the split call stream are used to negotiate the OOB connection, and are described in detail in Section 5. The FetchDataOOB call remains open during the transfer of the OOB file data. When the transfer of the file data is finished (whether successfully, or unsuccessfully), the FetchDataOOB call is terminated.

The abort codes are identical to those yielded by FetchData64.

### 4.2. StoreDataOOB

StoreDataOOB behaves very similarly to the extant RPC StoreData64. The RPC-L definition is:

```
StoreDataOOB(IN AFSFid *Fid, AFSStoreStatus *InStatus,
             afs_uint64 Pos, afs_uint64 Length,
             afs_uint64 FileLength,
             OUT AFSFetchStatus *OutStatus, AFSVolSync *Sync)
             split = XXX;
```

All of the arguments are identical to those of StoreData64. This is a split call, just like StoreData64, but the data in the split call stream is not the relevant file data. Instead, the contents of the split call stream are used to negotiate the OOB connection, and are described in detail in Section 5. The StoreDataOOB call remains open during the transfer of the OOB file data. When the transfer of the

file data is finished (whether successfully, or unsuccessfully), the StoreDataOOB call is terminated.

The abort codes are identical to those yielded by StoreData64.

## 5. Rx and OOB stream interaction

This section desribes the contents of the Rx split call stream for FetchDataOOB and StoreDataOOB. Note that the data over the stream is marshalled using XDR [RFC4506], but it is not encapsulated using the Record Marking Standard in [RFC5531] section 11, or anything else. It is just a single structure encoded in raw XDR by itself.

Note that this section only describes the use of a TCP-based protocol for the out-of-band data transport. Other transports may be defined by creating a new "version" of the AFSOOB_Challenge structure.

### 5.1. Rx split stream contents

The contents of the Rx split stream for both FetchDataOOB and StoreDataOOB are identical. The only data that appears on the split stream is a single AFSOOB_Challenge union, encoded in XDR, which is sent from the server to the client. This structure is specified as thus in RPC-L:

```
const AFSTCP_IPMAX = 128;
const AFSOOB_v1 = 1;

struct AFSTCP_IPv4 {
  afs_uint32 host;
  afs_uint16 port;
};

struct AFSOOB_v1_challenge {
  struct AFSTCP_IPv4 addrs<AFSTCP_IPMAX>;
};

union AFSOOB_Challenge switch (afs_uint32 type) {
  case AFSOOB_v1:
    struct AFSOOB_v1_challenge challenge;
};
```

AFSOOB_Challenge is a union, where the discriminant just dictates a logical "version" of the out-of-band challenge structure. Future revisions of this AFS-3 extension may involve more data in the challenge, which may be specified at a future date. Currently, only data for the AFSOOB_v1 challenge is defined, which is defined in the

AFSOOB_v1_challenge struct as follows:

addrs

This is an array containing pairs of IPv4 addresses and TCP ports that the client may attempt to connect to in order to initiate a TCP transfer. Several addresses MAY be specified, which SHOULD be attempted in order by the client. The "host" field indicates the IPv4 address to connect to. The server MAY specify an IPv4 address of 0.0.0.0, which indicates that the client MUST attempt to connect to the same IPv4 address that it is using for the associated RxUDP connection. The "port" field specifies the TCP port number to connect to.

## 5.2. XDR over TCP format

A few XDR-encoded structures are sent over the relevant TCP stream during a FetchDataTCP or StoreDataTCP call. Since the TCP stream is shared by both XDR-marshalled data and raw file data, in order to simplify processing, every logical block of XDR is prefixed by a length value. This "length" is a 32-bit unsigned integer, and is sent in network byte order over the TCP stream. Immediately following it is "length" number of octets representing the XDR stream. Anywhere in this document that refers to an XDR stream being transmitted over a TCP connection, this format MUST be used. Note that we do not use the Record Marking Standard from [RFC5531] section 11 for encapsulating XDR over TCP.

## 5.3. TCP stream negotiation

Once a client has received the challenge described in Section 5.1 over the relevant Rx split stream, the client opens a TCP connection to the fileserver (or the client MAY reuse an extant TCP connection, see Section 5.6). If the client succeeds in creating a connection, the client sends a response to the given challenge immediately on the TCP stream. The response consists of a single AFSTCP_Response union sent from the client to the server. The AFSTCP_Response union is defined as thus in RPC-L:

```
const AFSTCP_RXNULL = 0;
const AFSTCP_RXKAD = 2;

struct AFSTCP_v1_uniq {
  afs_uint32 host;
  afs_uint32 portAndServiceId;
  afs_uint32 epoch;
  afs_uint32 cid;
  afs_uint32 callNumber;
};

union AFSTCP_v1_private switch (afs_uint32 securityIndex) {
  case AFSTCP_RXNULL:
    void;
  case AFSTCP_RXKAD:
    struct AFSTCP_v1_rxkad encrypted;
};

struct AFSTCP_v1_response {
  struct AFSTCP_v1_uniq uniq;
  union AFSTCP_v1_private private;
};

union AFSTCP_Response switch (afs_uint32 type) {
  case AFSOOB_v1:
    struct AFSTCP_v1_response response;
};
```

AFSTCP_Response is a union, where the discriminant just dictates a logical "version" of the OOB out of band response structure. Future revisions of this AFS-3 extension may involve more data in the response, which may be specified at a future date. Currently, only data for the AFSOOB_v1 response is defined, which is defined in the AFSTCP_v1_response struct as follows:

uniq

The contents of this structure are defined immediately below.

private

This contains data specific to the security index used in the Rx connection of the associated Rx call. The discriminant is that security index, and it MUST match the security index of the Rx connection for the associated call. For unauthenticated connections (rxnull), this is an empty structure. For connections authenticated with rxkad, this is an AFSTCP_v1_rxkad structure, which is defined in

Section 5.4.

The AFSTCP_v1_uniq structure is used to identify which Rx call this connection is to be associated with. It is defined as follows:

host

The fileserver IPv4 address the client is using to connect to the relevant fileserver.

portAndServiceId

The least-significant 16 bits of this integer are the TCP port the client is using to connect to the relevant fileserver. The most-significant 16 bits are the service Id used in the relevant Rx call.

epoch

The Rx connection epoch for the associated connection.

cid

The connection Id for the associated Rx call channel and connection.

callNumber

The call number of the associated Rx call.

The client MUST fill this structure with the values matching the values in the corresponding Rx call. The server MUST verify that the values of the structure do correspond to an existing FetchDataTCP or StoreDataTCP Rx call, and that the given host and port are valid for the fileserver.

## 5.4. rxkad challenge response

The rxkad-specific portion of the challenge response structure transmitted over the TCP connection is described here. The entire structure is encrypted via fcrypt in ECB mode with the session key schedule and initialization vector of the Rx connection of the associated Rx call, and must be decrypted before its contents can be interpreted. The structure is defined in RPC-L as thus:

```
struct AFSTCP_v1_rxkad {
  struct AFSTCP_v1_uniq uniq;
  afs_uint32 seclevel;
```

```
};
```

After being decrypted, the structure contents are defined as follows:

uniq

: This MUST be identical to the uniquifier specified in the encompassing AFSTCP_v1_response structure, as defined in Section 5.3.

seclevel

: This corresponds to the "security level" that exists in rxkad. Currently, this field must always be 0 (rxkad_clear), and any other value MUST immediately cause an error to be raised. In the future, it may be possible to specify other security levels here which indicate that the file data should be encrypted according to the analogous rxkad security level, assuming the rxkad security index is specified.

If the uniquifier in this structure does not exactly match the uniquifier in the parent AFSTCP_v1_response structure, the TCP connection MUST be immediately closed and considered invalid. The invalid TCP connection SHOULD just be ignored, and thus the associated Rx call SHOULD NOT be aborted or terminated, and the server SHOULD continue to listen for TCP connections until a timeout is reached or the client aborts the Rx call. This is so unauthenticated attackers do not cause the Rx call to be aborted unnecessarily.

## 5.5. TCP stream payload

After the challenge response specified in Section 5.3 has been sent, the file data for the transfer will be transmitted. The file data is encoded as follows. The endpoint that is sending file data first sends a single XDR-encoded AFSTCP_FileData union (encoded according to Section 5.2), which is then followed by the raw file data itself. The AFSTCP_FileData union is defined as thus in RPC-L:

```
union AFSTCP_FileData switch (afs_uint32 type) {
  case AFSOOB_v1:
    afs_uint64 length;
};
```

AFSTCP_FileData is a union, where the discriminant just dictates a logical "version" of the TCP out of band file data structure. Future

revisions of this AFS-3 extension may involve more data being sent before the raw file data, which may be specified at a future date.

Currently, only data for the AFSOOB_v1 file data information is defined, which consists of a single field called "length". This field just dictates how many octets of raw file data will immediately follow the AFSTCP_FileData structure.

For a FetchDataTCP request, the server will immediately send the AFSTCP_FileData union across the TCP stream as soon as it has verified that the given challenge response is valid. For a StoreDataTCP request, the client will immediately send the AFSTCP_FileData union across the TCP stream without waiting for any response.

After the AFSTCP_FileData union has been sent, the raw file data immediately follows. Once all of the file data has been transmitted, the peer terminates the associated RXAFS call. The TCP connection MAY be terminated by the client, server, or both, after a successful transfer (see Section 5.6).

In the event of an error, there is no error code information transmitted over the TCP stream. If an error should occur, the peer that recognizes an error MUST immediately close the TCP stream, and send an Rx abort for the associated RXAFS call (unless otherwise specified, as in Section 5.4). The client may detect such an error by noticing that the TCP has been closed prematurely, and it receives the error code for the error from Rx abort codes from the associated Rx call. If the server notices a premature termination of the TCP connection, it MUST send an abort on the associated Rx call and stop processing.

## 5.6. TCP Connection Caching

Once a transfer has completed successfully, the TCP connection used does not need to be closed. The client and server MAY keep the connection open for use with a future transfer if they support the functionality in this section. A fileserver that supports this will, after a transfer is complete, treat the TCP connection as if it were a newly-accepted TCP connection. That is, the server will then wait for the connection to receive an AFSTCP_Response structure, or will time out the connection after an implementation-defined amount of time.

If the client wishes to make use of connection caching, it may also keep its end of the connection open, and may use that connection for a future transfer. All of the negotiation steps are the same as normal, with the sole exception that the client reuses the TCP connection instead of making a new one. Since all authentication negotiation occurs again, the cached TCP connections may be used for any user with any authentication credentials. However, the client

MUST ensure that the TCP is connected to an address that is valid according to the AFSTCP_Challenge received from the fileserver.

The caching of TCP connections is strictly OPTIONAL for both client and server. If either client or server does not support this, they may simply close the TCP connection after a successful transfer. If a client closes such a connection when the server supports caching, the server will take notice when listening for an AFSTCP_Response structure, and will just close its end of the connection. If the server closes such a connection when the client supports caching, the client will notice when it tries to reuse the connection; in which case, the client SHOULD simply discard the cached connection and create a new one.

## 6. Security Considerations

During negotiation of the TCP connection, the TCP stream is authenticated via fcrypt to prove that the connection is from the same user as the associated Rx call. It should be noted that fcrypt encryption is considered weak by modern cryptographic standards, and that this only authenticates the TCP stream itself. The contents of the TCP stream may be intercepted and read by an attacker, and the contents may even be modified by an attacker if they can successfully execute a TCP sequence number prediction attack.

These limitations may be addressed in the future by allowing different rxkad security levels or different Rx security indices to be specified by the client.

Clients that send an invalid AFSTCP_Response structure do not receive much useful feedback from the server in terms of error information or error codes. The Rx call MUST NOT be terminated due to TCP errors until the TCP stream is authenticated, or we risk a trivial DoS attack for the relevant Rx call. What happens is that the server just terminates the invalid TCP stream prematurely (which the client can notice), but it keeps the Rx call open until it times out, effectively waiting for the valid TCP connection.

It should also be noted that a sort of hijacking attack may be performed on anonymous transfers without needing to hijack any of the intermediate connections. An attacker that can eavesdrop on the relevant connections may detect the various Rx call and connection fields it needs to populate an AFSTCP_v1_uniq, and an attacker can then create its own TCP connection with the proper response, from any machine. This is not considered a problem, since data that may be

accessed anonymously is generally not sensitive (to some extent, it is not sensitive by definition). While it is possible that such data

may be protected by IP-based ACLs in AFS, such instances should be rare, and are already recommended against for other reasons. The server MAY alleviate this issue somewhat by restricting access based on the source IP address of the out-of-band TCP connection (for example, it may enforce that the source address of the TCP connection match the address of the Rx connection of the associated Rx call).

## 7. IANA Considerations

This document makes no request of the IANA.

## 8. AFS-3 Registry Considerations

This document requires the registration of two RPC code points in the RXAFS Rx package for the FetchDataTCP and StoreDataTCP RPCs detailed above in Section 4.

## 9. References

### 9.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC4506] Eisler, M., "XDR: External Data Representation Standard", STD 67, RFC 4506, May 2006.

### 9.2. Informative References

[AFS3-FSCM] Zayas, E., "AFS-3 Programmer's Reference: File Server/ Cache Manager Interface", Transarc Corp. Tech. Rep. FS-00-D162, August 1991.

[AFS3-RX] Zayas, E., "AFS-3 Programmer's Reference: Specification for the Rx Remote Procedure Call Facility", Transarc Corp. Tech. Rep. FS-00-D164, August 1991.

[RFC0959] Postel, J. and J. Reynolds, "File Transfer Protocol", STD 9, RFC 959, October 1985.

[RFC5531] Thurlow, R., "RPC: Remote Procedure Call Protocol Specification Version 2", RFC 5531, May 2009.

[RX] Zeldovich, N., "RX protocol specification".

## Appendix A. Acknowledgements

The author thanks Tom Keiser, Mike Meffie, and Mark Vitale for their discussion of the original AFS-3 OOB protocol design, on which the protocol described in this document is based.

## Author's Address

Andrew Deason
Sine Nomine Associates
43596 Blacksmith Square
Ashburn, Virginia 20147-4606
USA

Phone: +1 703 723 6673
Email: adeason@sinenomine.net