Modulo-N Incarnation Numbers for Cache-based Transport Protocols*

A. Udaya Shankar
Department of Computer Science and
Institute for Advanced Computer Studies
University of Maryland
College Park, Maryland 20742
shankar@cs.umd.edu

David Lee
AT&T Bell Laboratories
Murray Hill, New Jersey 07974
lee@research.att.com

Abstract

To provide reliable connection management, a transport protocol uses 3-way handshakes in which user incarnations are identified by bounded incarnation numbers from some modulo-N space. Recently, several cacheing schemes have been proposed to reduce the 3-way handshake to a 2-way handshake. In this paper, we define a class of cacheing protocols and determine the minimum value of N needed as a function of real-time constraints (e.g. message lifetime, incarnation creation rate, inactivity duration, cache residency times, etc.). The protocols use the client-server architecture and handle failures and recoveries. Both clients and servers generate incarnation numbers from a local counter (e.g. clock). These protocols assume a maximum duration for each incarnation; without this assumption, there is a very small probability ($\approx \frac{1}{N^2}$) of misinterpretation of incarnation numbers. This restriction can be overcome with some additional cacheing.

1 Introduction

The transport layer consists of clients and servers.[1] We assume that clients and servers can send messages to each other over channels that can lose, reorder and duplicate messages. (This is the typical network service available to the transport layer.) We also assume that clients, servers, and channels can fail.

*This work was supported in part by NSF Grant No. NCR 89-04590, and was done in part while consulting for AT&T Bell Laboratories.

[1] Actually, clients and servers are the users of the transport layer, and for each user there is a transport entity in the transport layer. But for notational brevity, we use "client" (and "server") to refer to both the user and the associated entity.

During the course of system execution, connections are opened and closed between clients and servers. A client can request a new connection to a server and close an established connection at any time. A server can either accept or reject an incoming connection request. We assume that between any client-server pair there is at most one connection at any time. This allows a client (server) to have multiple connections open at the same time, with different servers (clients).

This is a very general model, which subsumes, for example, the "well-known socket" architecture. Clients and servers typically represent user-level processes within hosts. But they may also represent hosts, with one client and server for each host.

Because the same client-server pair can undergo many connections over time, we need the notion of *incarnation* to precisely state the desired properties of connection management. A new incarnation of a client is started whenever the client requests a new connection to a server. A new incarnation of a server is started whenever the server accepts an incoming connection request. Once an incarnation is started, it has one of two possible futures (based on messages it receives): (1) it becomes *open to a remote incarnation*, and at some later point becomes *closed*; or (2) it becomes closed without ever becoming open.

A client incarnation closes without becoming open either because its connection request was rejected by the server or because of failure (in the server, the client, or the channels). A server incarnation closes without becoming open either because of failure or because it was started in response to a connection request that later turns out to be a duplicate request from some old (now closed) incarnation. Because of failures, it is also possible that an incarnation x becomes open to incarnation y but y becomes closed

0-8186-3670-X/93 $03.00 © 1993 IEEE

without becoming open.

A *connection* is an association between two open incarnations. Formally, a connection exists between incarnations x and y if y becomes open to x and x becomes open to y. The desired **correctness properties** for connection management are as follows:

- **Consistent connections**: If an incarnation x becomes open to an incarnation y, then incarnation y is either open to x or will become open to x unless there are failures. (Because an incarnation becomes open to at most one incarnation, this ensures "at-most-once" semantics; i.e. impossibility of two remote incarnations y and z that are both open to x.)
- **Consistent data-transfer**: If an incarnation x becomes open to an incarnation y, then x accepts received data only if sent by y.
- **Progress**: If a client incarnation x requests a connection to a server and the server does not reject x's request, then a connection is eventually established between x and an incarnation of the server.
- **Terminating handshakes**: It is not possible for a client-server pair to be in a state where each is sending messages to the other expecting a response that the other will never send. (Such "infinite chatter" is worse than deadlock because in addition to not making progress, the protocol is consuming precious network resources.)

The connection management function of traditional transport protocols (e.g. TCP) has two characteristics. First, successive incarnations are identified by increasing *incarnation numbers* from some modulo-N space. This means that the protocol must be designed to avoid **misinterpretable incarnation numbers**, i.e. an incarnation number of one incarnation being interpreted by an entity (client or server) as representing another incarnation.

Second, a server or client stores a remote incarnation's number only while it is connected to the remote incarnation. This necessitates a 3-way handshake for connection establishment [5, 8]. Consider a client that wants to connect to a server. The client sends a connection request with its incarnation number, say x. When the server receives this, it responds by sending a response containing x and a server incarnation number, say y. When the client receives the response, it becomes open to y and responds by sending an ack containing x and y. The server becomes open when it receives the ack. The server could not become open when it received the connection request containing only x (because it may have been an old duplicate).

Notice that if the server had remembered the incarnation number, say z, that the client previously used when it connected to the server, then the server could have distinguished that the connection request with x was new (because $x > z$). In that case, it could have become open at once; i.e. a 2-way handshake would suffice.

A server cannot be expected to remember the last incarnation number of every remote client to which it was connected, due to the enormous number of clients in a typical internetwork. However, a cacheing scheme is feasible, and several have been proposed recently (e.g. [4]), culminating in a proposed modification to TCP [2].

The motivation for reducing the number of handshakes comes from transaction-oriented users, such as RPCs. Notice that transaction data can be sent with a connection request, but the server cannot process the transaction until it confirms that this is a new request. Thus, with cacheing the delay can be reduced to the minimum possible: a roundtrip time plus server processing time.

There is an intricate relationship between the modulo-N space of the incarnation numbers and the handshaking algorithms. Most references in the literature seem to assume that misinterpretable incarnation numbers (often referred to as "wrap-around") are avoided if $N \geq \alpha \times \mathtt{L}$, where α is the minimum time between incarnation creations at an entity, and $\mathtt{L}$ is the maximum message lifetime imposed by the channels.

In fact, we show that this condition is not adequate. It ensures that messages in transit from different incarnations have different incarnation numbers. It allows the sender to correctly interpret the incarnation numbers in transit, because the sender knows the highest incarnation number, say z, that it has sent. But it does not ensure correct interpretation for the receiver, because the receiver only has an estimate of z. Furthermore, the receiver sends messages with its estimate of z to the sender, and the above condition does not ensure that the sender can correctly interpret these numbers either.

Our contribution

In this paper, we specify a class of cacheing protocols and obtain the minimum value of N that ensures the correctness properties. We assume that channels can lose, reorder and duplicate messages. We allow (fail-stop) failures and recoveries of clients, servers and channels.

In our protocols, each server caches the incarnation numbers of client incarnations that have connected to the server. Connection establishment is achieved in a 2-way (3-way) handshake if an entry for the requesting client is (is not) found in the cache. Connection closing and connection request rejection is achieved by a 2-way handshake. In addition to data transfer with the connection establishment phase, there is also a data transfer phase which can use any of the typical data-transfer mechanisms (e.g. sliding window).

Each entity (client and server) has a maximum "wait" duration. If a response is outstanding for longer than that duration, it assumes failure (of the remote entity or channels) and aborts the connection.[2] When a failed client recovers, it can request a new connection immediately. Thus it is possible for a server connected to client to receive a connection request from the client with a higher incarnation number (if the client failed and recovered before the server's wait duration elapsed). In that case, the server closes the current incarnation and (optionally) can start a new connection with a new incarnation. The same can happen to the client: if the server fails, recovers, and receives an old duplicate connection request, then the server will respond to this request.

Our protocols can accommodate any size of the server cache, including no cache at all. However, if an entry is cached, then it must be cached for a minimum time (otherwise correctness can be violated); it can be flushed out any time after that. We allow the cache to be lost in a crash.

Ideally, a server's cache should be large enough to store the incarnation number of every client that has connected to the server for a period of the maximum message lifetime plus client wait duration. In this case, the 3-way handshake can be eliminated entirely.

In our class of protocols, denoted SC (for Server-Cache), each entity obtains incarnation numbers from an *incarnation number generator*, which supplies increasing (but not necessarily consecutive) numbers to successive incarnations. Thus, the generator can be a counter or a real-time clock. We assume it is not lost in a crash.

Organization of paper

In Section 2, we specify a protocol SC1, which assumes unbounded incarnation numbers, and prove that it satisfies the correctness conditions. In Section 3, we show that SC1 can be modified to use modulo-N incarnation numbers, resulting in protocol SC2. Section 4 contains concluding remarks. All the proofs are omitted for space reasons; they can be found in [7].

The following notation is used throughout the paper:

- α is the minimum time between successive incarnation generations.
- $\mathtt{W_C}$ ($\mathtt{W_S}$) is the maximum wait duration for the client (server).
- $\mathtt{L}$ is the maximum message lifetime in a channel.
- $\mathtt{C_S}$ is the maximum duration of an entry in the server cache; we assume $\mathtt{C_S} > \mathtt{L} + \mathtt{W_C}$.
- $\mathtt{I}$ is the maximum duration of an incarnation.

2 Protocol SC1: Unbounded Incarnation Numbers

Every entity (client or server) has a local incarnation number generator, which supplies increasing numbers for identifying incarnations. Successive incarnation numbers do not have to be consecutive. Thus, the generator can be a counter or a real-time clock. We assume that this generator is not lost if the entity crashes. That is, when the entity recovers, the generator is reinitialized to the previous value (if it is a counter) or to the current time (if it is a clock).

Consider a client-server pair. The server caches an incarnation number x in two ways: (1) if it becomes open to x as a result of a 3-way handshake; and (2) if it receives a connection request with incarnation number x when its cache has a value $z < x$ (whether or not the server accepts the request). In both cases, the cacheing of x signifies that the server has not connected previously to incarnation x or to any later incarnation of the client.

The server does not need to remember the value x after $\mathtt{L} + \mathtt{W_C}$ seconds since cacheing x. This is because any connection request received after that time comes from a later incarnation to which the server can open at once[3]. Thus after this time, the server just needs to remember that $\mathtt{L} + \mathtt{W_C}$ seconds have elapsed since the cache was updated. Note that it must remember this much; otherwise it would not be able to distinguish this situation from a post-crash period when the cache is lost and a 3-way handshake is needed.

Convention: Throughout, we use a to range over

[2] Thus, every opening and closing interval is bounded by this duration, as also is any open period where a data ack is outstanding.

[3] $\mathtt{W_C}$ seconds after the server caches x, the client no longer sends any connection request with incarnation number x; after another L seconds, no such connection request would be in the channels.

client ids and b to range over server ids.

Each client a maintains the following state variables:

$LinGen_a$: $\{0, 1, \cdots\}$. Local incarnation number generator. Initially 0.

$Status_a(b)$: {**closed**, **opening**, **open**, **closing**}. Initially **closed**.
Status of client's relationship with server b. **closed** iff client has no incarnation involved with b. **opening** means client has an incarnation requesting a connection with b. **open** means client has an incarnation open to b. **closing** means client has an incarnation closing a connection with b.

$Lin_a(b)$: $\{\texttt{nil}\} \cup \{0, 1, \cdots\}$. Initially **nil**.
Local incarnation number. **nil** if $Status_a(b)$ = **closed**. Otherwise identifies client incarnation involved with server b.

$Din_a(b)$: $\{\texttt{nil}\} \cup \{0, 1, \cdots\}$. Initially **nil**.
Distant incarnation number. **nil** if $Status_a(b)$ equals **closed** or **opening**. Otherwise identifies the incarnation of server b with which the client incarnation is involved.

Each server b maintains the following state variables:

$LinGen_b$: $\{0, 1, \cdots\}$. Local incarnation number generator. Initially 0.

$Status_b(a)$: {**closed**, **opening**, **open**}. Initially **closed**.
Status of server's relationship with client a. **closed** iff server has no incarnation involved with a. **opening** means server has an incarnation accepting a connection request from a. **open** means server has an incarnation open to a.

$Lin_b(a)$: $\{\texttt{nil}\} \cup \{0, 1, \cdots\}$. Initially **nil**.
Local incarnation number. **nil** if $Status_b(a)$ = **closed**. Otherwise identifies server incarnation involved with client a.

$Din_b(a)$: $\{\texttt{nil}\} \cup \{0, 1, \cdots\}$. Initially **nil**.
Distant incarnation number. **nil** if $Status_b(a)$ = **closed**. Otherwise identifies the incarnation of client a with which the server incarnation is involved.

$Cache_b(a)$: $\{\texttt{nil}, \texttt{old}\} \cup \{0, 1, \cdots\}$. Initially **nil**.
Cache entry for client a. $Cache_b(a)$ = **nil** indicates that cache contains no entry for client a. $Cache_b(a)$ = **old** indicates that at least $\mathtt{L} + \mathtt{W_C}$ seconds have elapsed since last update of cache entry for client a. If $Cache_b(a) \notin \{\texttt{nil}, \texttt{old}\}$, then it indicates the (cached) incarnation number used by client a when it last connected (or attempted to connect) to server; furthermore this occurred less than $\mathtt{L} + \mathtt{W_C}$ seconds ago.

Note that if the server is open and $Cache_b(a) \neq$ **old**, then $Cache_b(a) = Din_b(a) \neq$ **nil**.

We next describe the messages exchanged between clients and servers. Each message is of the form (**M**, *sid*, *rid*, *sin*, *rin*), where **M** is the type of the message, *sid* is the sender's id, *rid* is the intended receiver's id, *sin* is the sender's incarnation number, and *rin* is the intended receiver's incarnation number. In some messages, *sin* or *rin* may be absent. For notational brevity, we have omitted the optional data fields in messages, as also messages related to the data transfer phase.[4] (Concerning the analysis of misinterpretable incarnation numbers, data transfer messages are equivalent to the **DR** and **DRACK** messages defined below.)

Each message is either a *primary* message or a *secondary* message. A primary message is sent repeatedly[5] until a response is received. A secondary message is sent only in response to the reception of a primary message. Note that the response to a primary message may be another primary message (as in a 3-way handshake).

We next list the messages sent by clients:

(**CR**, *sid*, *rid*, *sin*). Connection request. Sent when opening. Primary message.

(**CRRACK**, *sid*, *rid*, *sin*, *rin*). Acknowledgement to connection request reply. Secondary message.

(**DR**, *sid*, *rid*, *sin*, *rin*). Disconnect request. Sent when closing. Primary message.

(**REJ**, *sid*, *rid*, *rin*). Reject response to a connection request reply (**CRR**) which is received when closed. The *sin* of the received **CRR** is used as the value of *rin*. Secondary message.

The messages sent by servers are as follows:

(**CRR**, *sid*, *rid*, *sin*, *rin*). Reply to connection request in 3-way handshake. Sent when opening. Primary message.

(**CRACK**, *sid*, *rid*, *sin*, *rin*). Acknowledgement to connection request in 2-way handshake. Sent if cache has entry for *sid*. Secondary message.

(**DRACK**, *sid*, *rid*, *sin*, *rin*). Response to disconnect request. Secondary message.

(**REJ**, *sid*, *rid*, *rin*). Reject response to a **CR** received when closed. The *sin* of the received message is used

[4] Such messages contain all the fields mentioned above and additional fields such as sliding window sequence numbers and size. It is trivial to add the data transfer function to the connection management protocol defined here [6].

[5] according to some retransmission policy.

as the value of rin. Secondary message.

Figures 1 and 2 illustrate connection establishment by 3-way and 2-way handshakes. Figures 3 and 4 illustrate connection rejection.

The events of client a are shown in Figure 7, and the events of server b are shown in Figure 8. There are two types of events. A "nonreceive" event has an enabling condition (ec) and an action (ac); the action can be executed whenever the event is enabled. A receive event for a message has only an action; it is executed whenever the message is received. We assume that $LinGen$ is incremented at least once between successive reads (by an event not shown). We use abbreviations like $sin > Cache_b(a) \notin \{\texttt{nil}, \texttt{old}\}$ to denote $Cache_b(a) \notin \{\texttt{nil}, \texttt{old}\} \wedge sin > Cache_b(a)$.

Failure model: We assume that an entity (client or server) can fail and recover at any time. It executes no events while failed. Upon recovery, it reinitializes the *Status*, *Lin* and *Din* for every remote entity. We assume that $LinGen$ is not lost in a failure. The server cache may be lost.

Theorem 1. Protocol SC1 satisfies the correctness properties of consistent connections, consistent data-transfer, progress, and terminating handshakes, assuming the following:

(T1) $\mathtt{W_C} < \min(c_S, r_S)$ and $\mathtt{W_S} < \min(w_C, r_C)$

where w_C (w_S) is the minimum wait duration for the client (server), c_S is the minimum duration of an entry in the server cache, and r_C (r_S) is the minimum recovery time for the client (server).

Figures 5 and 6 show how the consistent-connections property can be falsified if **T1** does not hold.

3 Protocol SC2: Modulo-N Incarnation Numbers

We next determine conditions under which we can replace the unbounded incarnation numbers by modulo-N values, for some N.

To illustrate the approach, let's consider the sin numbers from client a received at server b. In protocol SC1, if the server receives a sin value, it tests sin against $Din_b(a)$; there are two types of tests, $sin = Din_b(a)$ and $sin > Din_b(a)$ (depending on the message received and the state of the server). If we replace sin by $sin \bmod N$, we must also replace each test by an equivalent test that reaches the same conclusion.

We can do this if sin is within fixed bounds of $Din_b(a)$, i.e. if $sin \in [Din_b(a) - K_1, \cdots, Din_b(a) + K_2]$ for some K_1 and K_2. In this case, we can replace sin by $sin \bmod N$ provided $N \geq K_1 + K_2$. Then the test $sin = Din_b(a)$ becomes the test $sin = Din_b(a) \bmod N$. The test $sin > Din_b(a)$ becomes $sin \in [(Din_b(a) + 1) \bmod N, \cdots, (Din_b(a) + K_2) \bmod N]$, which can be computed efficiently as $1 \leq sin \ominus Din_b(a) \leq K_2$, where $\ominus$ is modulo-N subtraction.

The same treatment is needed for the rin numbers, which are compared against the server's $Lin_b(a)$. And the same has to be done at the client side. In short, we need to determine conditions under which we can bound the sin and rin numbers received by an entity relative to the entity's Din and Lin variables.

We have established (in Theorem 2 of [7]) that the following bound on N suffices:

(T2) $N \times \alpha \geq 2\mathtt{L} + \mathtt{W_S} + \max(2\mathtt{W_C} + \mathtt{C_S}, 2\mathtt{L} + 2\mathtt{W_C} + \mathtt{W_S}, 2\mathtt{L} + \mathtt{W_S} + \mathtt{I})$

We obtain protocol SC2 by modifying SC1 as follows:

- Redefine the domains of variables $LinGen_a$, $Lin_a(b)$, $Din_a(b)$, $LinGen_b$, $Lin_b(a)$, $Din_b(a)$, $Cache_b(a)$, and message fields sin and rin to be $\{0, \ldots, N-1\}$
- Every test of *equality* involving these variables and fields (e.g. $sin = Din_b(a)$) is unchanged (but now each side is a modulo-N number).
- Replace the test $sin > Cache_b(a)$ in the server when closed or open by $1 \leq sin \ominus Cache_b(a) \leq \frac{\mathtt{L}+\mathtt{W_C}+\mathtt{C_S}+\mathtt{W_S}}{\alpha}$
- Replace the test $sin > Din_b(a)$ in the server when opening by $1 \leq sin \ominus Din_b(a) \leq \frac{\mathtt{L}+\mathtt{W_C}+\mathtt{W_S}}{\alpha}$
- Replace the test $sin > Din_a(b)$ in the client when open by $1 \leq sin \ominus Din_a(b) \leq \frac{2\mathtt{L}+\mathtt{W_C}+\mathtt{W_S}}{\alpha}$

Theorem 3. Protocol SC2 satisfies the correctness properties if N satisfies **T2**.

Almost always, $\mathtt{I}$ is much greater than $\mathtt{W_C}$ and $\mathtt{W_S}$. Then the bound **T2** approximates to

$$N \times \alpha \geq 2\mathtt{L} + \max(\mathtt{C_S}, \mathtt{I})$$

Typically, $\mathtt{I}$ is also much greater than $\mathtt{L}$, and the above bound simplifies to $N \times \alpha \geq \mathtt{I}$.

For example, if we use 32-bit incarnation numbers ($N = 2^{32}$) and assume a maximum incarnation generation rate of 10^4 incarnations per second, then the above bound requires incarnation lifetimes to be less than 100 hours.

The only drawback of protocol SC2 is N's dependence on $\mathtt{I}$ (TCP suffers from this too). This happens through the DR and DRACK messages [7]. In each of these messages, when an entity receives the message

it tests for $sin = Din \wedge rin = Lin$. Thus, misinterpretation can occur only if client and server incarnation numbers in the current connection are exactly the same as client and server incarnation numbers in the previous (long-lived) connection. The probability of this is very low; specifically, it equals $\frac{1}{N^2}$, assuming that incarnation start times are distributed uniformly at random.

If we are willing to live with this probability of misinterpretation, then we get the following lower bound on N (by ignoring the I constraint):

(T2') $N \times \alpha \geq 2\mathrm{L} + 2\mathtt{W_C} + \mathtt{W_S} + \max(\mathtt{C_S}, 2\mathrm{L} + \mathtt{W_S})$

For those situations where this probability is not negligible, we can completely eliminate the I constraint by using an additional cache, referred to as a *Lin-generator cache*, at either the server or the client (or both). The entity with the generator cache, say a, stores its own incarnation number from its previous connection with the other entity, say b, for at least 2L seconds. When a is next involved in a connection (or connection attempt) with b, if its generator cache contains an entry for b, it uses an incarnation number one higher than the entry; otherwise, it uses an arbitrary incarnation number. The generator cache must not be lost in crash, unlike the usual server cache. Given such a generator cache, **T2'** ensures correct interpretation of incarnation numbers, and hence correct operation [7].

Finally, protocol SC2 can completely avoid 3-way handshakes by satisfying the following additional constraints:

(T3) Each cache entry is stored for at least $\mathrm{L} + \mathtt{W_C}$ seconds, and no cache entry is lost due to crash.

This ensures that $Cache_b(a)$ is never `nil`; hence there are no 3-way handshakes, **opening** state, or CRR messages.

4 Conclusions

We have presented a transport protocol that uses server cacheing to achieve 2-way connection establishment, suitable for transaction-oriented users such as RPCs. When no cache entry is available, the protocol resorts to 3-way handshakes. By having a sufficiently large cache, 3-way handshakes can be eliminated entirely. Our protocols tolerate crashes, and require entities to wait for only a brief delay upon recovery.

Our protocol uses modulo-N numbers to identify incarnations. We have obtained the minimum value of N that guarantees correct operation. To our knowledge, no such bound has been previously presented in the general setting we have considered.

Unlike other cacheing protocols proposed [2], our protocol does not use a cache entry if it is older than the maximum message lifetime plus maximum client wait duration. This is key to ensuring that even if N does not satisfy the lower bound with respect to maximum incarnation lifetime, the probability of misinterpretation is very low.

Timer-based techniques provide another approach for achieving connection-establishment with 2-way handshakes [1, 9, 3]. These techniques are conceptually simpler, but they require clients and servers to have accurately synchronized clocks. Maintaining the desired synchronization can be difficult in a heterogenous wide-area environment such as the Internet.

References

[1] E. W. Biersack and D. C. Feldmeier. A Timer-Based Connection Management Protocol with Synchronized Clocks and its Verification. Technical report, Bellcore, Morristown, NJ 07962, July 1992.

[2] R. Braden. Extending TCP for Transactions – Concepts. Request for Comment RFC-1379, Network Information Center, November 1992.

[3] J. G. Fletcher and R. W. Watson. Mechanisms for a Reliable Timer Based Protocol. *Computer Networks*, 2(4/5):271–290, September 1978.

[4] L. Garlick, R. Rom, and J. Postel. Issues in Reliable Host-to-Host Protocols. In *Second Berkeley Workshop on Distributed Data Management and Computer Networks*, May 1977.

[5] J. Postel. Transmission Control Protocol: DARPA Internet Program Protocol Specification. Request for Comment RFC-793, STD-007, Network Information Center, September 1981.

[6] A.U. Shankar. Modular Design Principles for Protocols with an Application to the Transport Layer. *Proceedings of the IEEE*, December 1991.

[7] A.U. Shankar and D. Lee. Modulo-N Incarnation Numbers for Cache-based Transport Protocols. Technical Report CS-TR-3046, Computer Science Department, University of Maryland, March 1993. Also AT&T Bell Labs 11272-930520-19TM.

[8] C.A. Sunshine and Y.K. Dalal. Connection Management in Transport Protocols. *Computer Networks*, 2(6), December 1978.

[9] R. W. Watson. The Delta-t Transport Protocol: Features and Experience. In *IEEE 14th Conferance on Local Computer Networks*, pages 399–407, Minneapolis, MN, October 1989.

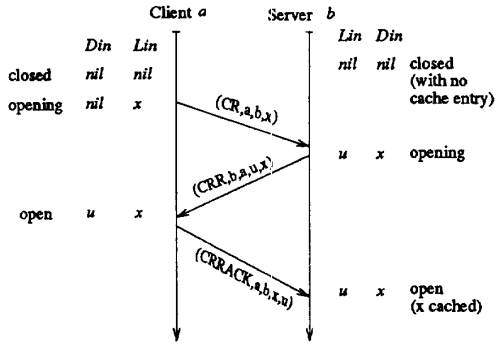

Figure 1: 3-way connection establishment.

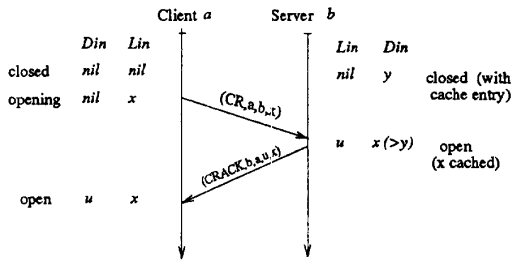

Figure 2: 2-way connection establishment.

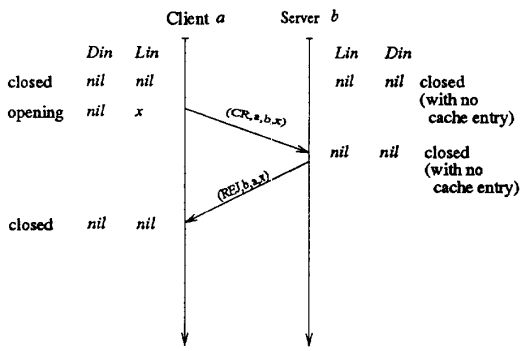

Figure 3: Connection rejection when no cache entry.

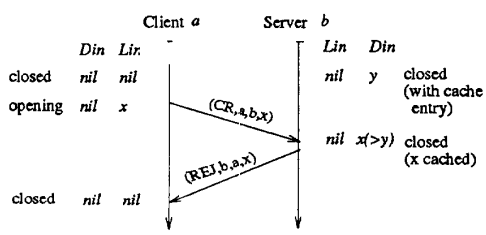

Figure 4: Connection rejection when cache entry present.

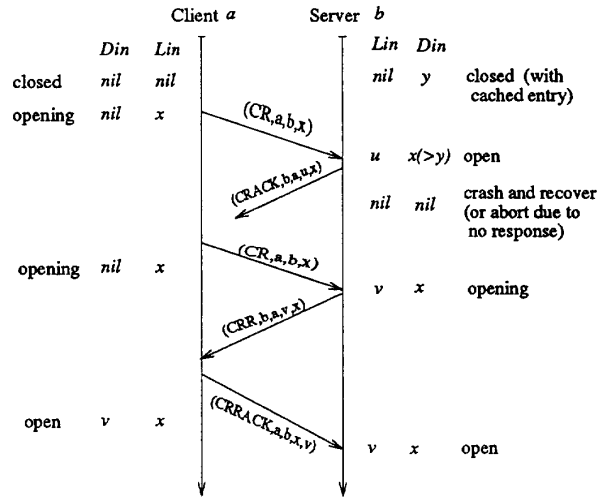

Figure 5: Scenario where incarnations u and v of server b become open to incarnation x of client a.

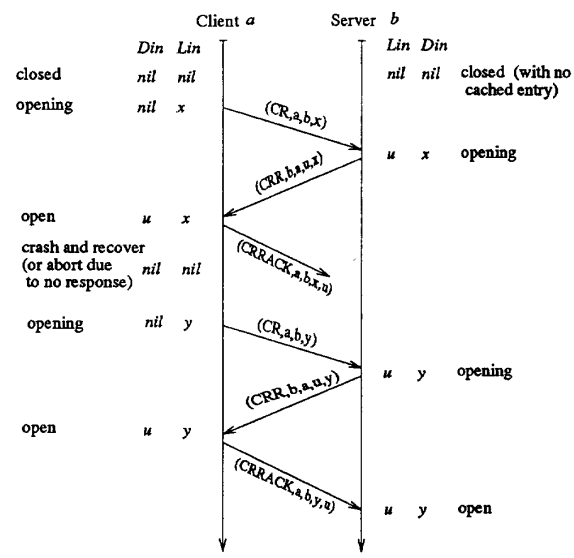

Figure 6: Scenario where incarnations x and y of client a become open to incarnation u of server b.

Events of client a concerning server b

$ConnectRequest_a(b)$
- ec: $Status_a(b) = \texttt{closed}$
- ac: $Status_a(b) := \texttt{opening};\ Lin_a(b) := LinGen_a$

$DisconnectRequest_a(b)$
- ec: $Status_a(b) = \texttt{open}$
- ac: $Status_a(b) := \texttt{closing}$

$Abort_a(b)$
- ec: $Status_a(b) \neq \texttt{closed} \wedge$ ⟨response outstanding for more than $\texttt{W}_\texttt{C}$ seconds⟩
- ac: $Status_a(b) := \texttt{closed};\ Lin_a(b) := \texttt{nil};\ Din_a(b) := \texttt{nil}$

$Send\texttt{CR}_a(b)$
- ec: $Status_a(b) = \texttt{opening}$
- ac: $Send(\texttt{CR}, a, b, Lin_a(b))$

$Send\texttt{DR}_a(b)$
- ec: $Status_a(b) = \texttt{closing}$
- ac: $Send(\texttt{DR}, a, b, Lin_a(b), Din_a(b))$

$Receive(\texttt{CRR}, b, a, sin, rin)$
- ac: $Status_a(b) = \texttt{opening} \wedge rin = Lin_a(b) \longrightarrow$ {3-way handshake}
  $Status_a(b) := \texttt{open};\ Din_a(b) := sin;$
  $Send(\texttt{CRRACK}, a, b, Lin_a(b), Din_a(b))$
- □ $Status_a(b) = \texttt{open} \wedge rin = Lin_a(b) \wedge sin = Din_a(b) \longrightarrow$ {duplicate CR}
  $Send(\texttt{CRRACK}, a, b, Lin_a(b), Din_a(b))$
- □ $Status_a(b) = \texttt{open} \wedge rin = Lin_a(b) \wedge sin > Din_a(b) \longrightarrow$ {server crashed, recovered, responding to old CR}
  $Send(\texttt{REJ}, a, b, sin);\ Status_a(b) := \texttt{closed};$
  $Din_a(b) := \texttt{nil};\ Lin_a(b) := \texttt{nil}$
- □ $Status_a(b) \in \{\texttt{closed}, \texttt{closing}\} \longrightarrow Send(\texttt{REJ}, a, b, sin)$

$Receive(\texttt{CRACK}, b, a, sin, rin)$
- ac: $Status_a(b) = \texttt{opening} \wedge rin = Lin_a(b) \longrightarrow$ {2-way handshake}
  $Status_a(b) := \texttt{open};\ Din_a(b) := sin;$
- □ $Status_a(b) \in \{\texttt{open}, \texttt{closing}, \texttt{closed}\} \longrightarrow$ ⟨no action⟩

$Receive(\texttt{REJ}, b, a, rin)$
- ac: $Status_a(b) \in \{\texttt{opening}, \texttt{closing}\} \wedge rin = Lin_a(b) \longrightarrow$
  $Status_a(b) := \texttt{closed};\ Din_a(b) := \texttt{nil};\ Lin_a(b) := \texttt{nil}$
- □ $Status_a(b) \in \{\texttt{open}, \texttt{closed}\} \longrightarrow$ ⟨no action⟩

$Receive(\texttt{DRACK}, b, a, sin, rin)$
- ac: $Status_a(b) = \texttt{closing} \wedge rin = Lin_a(b) \wedge sin = Din_a(b) \longrightarrow$
  $Status_a(b) := \texttt{closed};\ Din_a(b) := \texttt{nil};\ Lin_a(b) := \texttt{nil}$
- □ $Status_a(b) \in \{\texttt{opening}, \texttt{open}, \texttt{closed}\} \longrightarrow$ ⟨no action⟩

Figure 7: Events of Client a in Protocol SC1.

Events of server b concerning client a

$MakeOldCache_b(a)$ {executed between $\mathtt{L} + \mathtt{W_C}$ and $\mathtt{C_S}$ seconds of becoming enabled}

ec: $Cache_b(a) \notin \{\mathtt{nil}, \mathtt{old}\}$

ac: $Din_b(a) := \mathtt{old}$

$Abort_b(a)$

ec: $Status_b(a) \neq \mathtt{closed} \wedge$ ⟨response outstanding for more than $\mathtt{W_S}$ seconds⟩

ac: $Status_b(a) := \mathtt{closed}$; $Lin_b(a) := \mathtt{nil}$

$Send\mathtt{CRR}_b(a)$

ec: $Status_a(b) = \mathtt{opening}$

ac: $Send(\mathtt{CRR}, b, a, Lin_b(a), Din_b(a))$

$Receive(\mathtt{CR}, a, b, sin)$

ac: $Status_b(a) = \mathtt{closed} \wedge$ ⟨rejecting connnections⟩ $\longrightarrow$
$Send(\mathtt{REJ}, b, a, sin)$;
if $sin > Cache_b(a) \neq \mathtt{nil}$ then $Cache_b(a) := sin$

□ $Status_b(a) = \mathtt{closed} \wedge$ ⟨ accept conn⟩ $\wedge$ $Cache_b(a) = \mathtt{nil}$ $\longrightarrow$ {no cache entry
$Status_b(a) := \mathtt{opening}$; $Lin_b(a) := LinGen_b$; $Din_b(a) := sin$ 3-way hndshk}

□ $Status_b(a) = \mathtt{closed} \wedge$ ⟨ accept conn⟩
$\wedge$ ($Cache_b(a) = \mathtt{old} \vee sin > Cache_b(a) \neq \mathtt{nil}$ $\longrightarrow$ {cache entry, 2-way hndshk}
$Status_b(a) := \mathtt{open}$; $Lin_b(a) := LinGen_b$; $Din_b(a) := sin$;
$Cache_b(a) := sin$; $Send(\mathtt{CRACK}, b, a, Lin_b(a), Din_b(a))$

□ $Status_b(a) = \mathtt{opening} \wedge sin > Din_b(a)$ $\longrightarrow$ {previous $Din_b(a)$ value
$Din_b(a) := sin$ was from some old CR}

□ $Status_b(a) = \mathtt{open}$
$\wedge$ ($Cache_b(a) = \mathtt{old} \vee sin > Cache_b(a) \notin \{\mathtt{nil}, \mathtt{old}\}$) $\longrightarrow$ {client crashed, reconnecting}
if ⟨willing to reopen⟩ then
$Lin_b(a) := LinGen_b$; $Din_b(a) := sin$; $Cache_b(a) := sin$;
$Send(\mathtt{CRACK}, b, a, Lin_b(a), Din_b(a))$
else $Status_b(a) := \mathtt{closed}$; $Lin_b(a) := \mathtt{nil}$; $Din_b(a) := sin$; $Cache_b(a) := sin$

□ $Status_b(a) = \mathtt{open} \wedge sin = Cache_b(a) \notin \{\mathtt{nil}, \mathtt{old}\}$ $\longrightarrow$ {duplicate CR}
$Send(\mathtt{CRACK}, b, a, Lin_b(a), Din_b(a))$

$Receive(\mathtt{CRRACK}, a, b, sin, , rin,)$

ac: $Status_b(a) = \mathtt{opening} \wedge sin = Din_b(a) \wedge rin = Lin_b(a)$ $\longrightarrow$
$Status_b(a) := \mathtt{open}$; $Cache_b(a) := Din_b(a)$

□ $Status_b(a) \in \{\mathtt{open}, \mathtt{closed}\}$ $\longrightarrow$ ⟨no action⟩

$Receive(\mathtt{DR}, a, b, sin, rin)$

ac: $Status_b(a) = \mathtt{open} \wedge sin = Din_b(a) \wedge rin = Lin_b(a)$ $\longrightarrow$
$Send(\mathtt{DRACK}, b, a, Lin_b(a), Din_b(a))$;
$Status_b(a) := \mathtt{closed}$; $Lin_b(a) := \mathtt{nil}$; $Din_b(a) := \mathtt{nil}$

□ $Status_b(a) = \mathtt{closed}$ $\longrightarrow$ $Send(\mathtt{DRACK}, b, a, rin, sin)$;

□ $Status_b(a) = \mathtt{opening}$ $\longrightarrow$ ⟨no action⟩

$Receive(\mathtt{REJ}, a, b, rin)$

ac: $Status_b(a) = \mathtt{opening} \wedge rin = Lin_b(a)$ $\longrightarrow$ $Status_b(a) := \mathtt{closed}$; $Lin_b(a) := \mathtt{nil}$;

□ $Status_b(a) \in \{\mathtt{open}, \mathtt{closed}\}$ $\longrightarrow$ ⟨no action⟩

Figure 8: Events of Server b in Protocol SC1.