Reliable Message Delivery and Conditionally-Fast Transactions are not Possible without Accurate Clocks

Mark A. Smith *

AT&T Labs Research
180 Park Ave., Florham Park, NJ 07932
mass@research.att.com

Abstract

In this paper we examine reliable transport level protocols for efficient transactions across a network. A typical transaction is a request from a client and a response from a server. The canonical example being remote procedure call. Transport level protocols such as TCP [14] and ISO TP-4 [9] work well for data streaming, but are inefficient for transactions. However, several protocols [19, 11, 17] have been designed to meet both needs, including a proposed extension of TCP call T/TCP [4, 5]. The goal of T/TCP is not to perform efficient transactions all the time, but only under certain conditions. However, in examining T/TCP [18] we observed that in certain situations the protocol may deliver the same message twice, even when efficient transactions are not required. This observation lead us to consider whether any protocol can deliver streams of data reliablely and still have fast transactions under the same conditions required by T/TCP. We present a formal definition of what it means to provide both services under the conditions proposed by the designers of T/TCP, and prove that without "accurate" clocks, it is impossible for any protocol to solve this problem. We also present a precise formal model that we use to describe the system and present the proofs. The model is a novel combination of a model with liveness properties and a model that allows local clocks.

1 Introduction

TCP/IP transport level protocols are responsible for reliable delivery of data between users that are typically application programs such as ftp, telnet, or email. On the Internet packets sent from one user application to another may get duplicated, lost, or arrive out of order. Reliable transport level protocols like TCP [14] and ISO TP-4 [9] are designed to ensure that the application programs receive messages[1] without duplication, without loss, and in the correct order. While these protocols work well for data streaming, they do not work well for transactions because they use a connection management mechanism that forces two round trips across the network for a client application to send a request and get a response from a server application. Ideally the request and response should be done in one round trip across the network. In order to have transactions in one round-trip across the network, a one way trip across the network must be sufficient for the server to deliver a message from the client. TCP and ISO TP-4 rely on unique identifiers (UID's) and a three-way handshake protocol to establish connection and ensure reliable message delivery. That is, to send a message the client first generates a UID x which it sends to the server; when the server receives this UID, it generates a UID y and sends back $\langle x, y \rangle$ to the client; the client then sends the request message together with y. In this way the server knows the message is not a duplicate. The server can then send the response with either x or y.

In order to ensure reliable delivery, hosts maintain some state information for each incarnation of a connection. However, in typical network situations, client and server hosts may have many different connections in parallel. Additionally, there may be different *incarnations* of the same connection, as the connection is opened closed and then opened again. Therefore, because of the number of connections a host may be involved with, this state information cannot be maintained forever. Therefore, hosts will periodically quiesce, that is, delete state information associated with a connection. In TCP whenever a connection is closed the state information associated with that connection is deleted. However, if the server does not immediately quiesce when a connection closes then *timer-based* mechanisms can be used for reliable transport level protocols. For example, Watson's Delta-t protocol [19] relies on clocks that run at the rate of real time and exploits the knowledge of the maximum packet lifetime (MPL) to achieve transactions in one round trip across the network. For the Delta-t protocol, quiesce time is based on the MPL. If the client and server hosts are assumed to have approximately synchronized clocks, then the protocol by Liskov, Shrira, and Wroclawski [11] also only requires one round trip

*Much of this work was done while the author was at the M.I.T. Lab for Computer Science, and was supported by ARPA contract F19628-95-C-0118.

PODC 98 Puerto Vallarta Mexico

[1] We use the term "message" or "data" for user-meaningful data and the term "packet" to denote objects sent over the channels by a protocol.

across the network for transactions, and quiesce time depends on the message delivery time.

Braden and Clark [4, 5] have also designed a protocol to achieve efficient transactions and work well for data streaming. Their protocol does not rely on approximately synchronized clocks or strict enforcement of MPL. Their approach is based on the idea that some information related to incarnations can be stored indefinitely and efficiently in caches when a connection closes, and that the protocol while ensuring efficient transactions most of the time, is allowed to be inefficient in some situations — typically after crashes. Their protocol is an extension of TCP, and because it is designed to perform transactions efficiently, it is called T/TCP. In most circumstances T/TCP can complete a transaction in one round trip across the network. However, it reverts to two round trips if the cached state is inappropriate. In [17] Shankar and Lee design a class of "caching" protocols that are similar to T/TCP. However, the class of protocols they design make explicit use of MPL and other timing information. In studying T/TCP [18] we noticed that under certain circumstance T/TCP may deliver the same message twice if timing information is not used. This observation lead us to consider whether it is possible for any protocol to perform transactions efficiently under the same circumstances in which T/TCP is required to perform efficient transactions and still deliver messages reliablely.

1.1 Our work

In this paper we present a formal definition of the correctness requirements for protocols that are allowed to keep data between incarnations indefinitely and allow inefficient transactions sometimes, but must always deliver data reliably. Our formal definition of the problem is based on the requirements for T/TCP. Let d be the maximum time it takes for a packet to traverse the network in an execution of the system. We prove that in a system where the client and the server have UID's and local clocks, that even if we require fast delivery only when the processes have appropriate state information, the clocks are accurate, and packets are not lost, then any protocol that takes time less than $2d$ for the server to deliver a message from the client may not deliver data reliably if the local clocks may sometimes run at arbitrary rates. If there is no maximum packet lifetime, then we prove that if the client and server have clocks that run at the rate of real time, but whose values maybe shifted by some arbitrary amount, again it is impossible to solve the problem without using a delivery time of at least $2d$. The $2d$ bound means that it is impossible for any protocol to complete transactions in one round trip across the network and still deliver messages reliably when the above mentioned timing uncertainties exist, even when fast transactions are only required under somewhat ideal conditions.

1.2 Related work

In addition to the practical work mentioned above, there has also been significant theoretical work in the study of reliable message delivery protocols. The earlier work in the area considered just the possibility of reliable message delivery and mostly in a purely asynchronous setting. This is the case for the impossibility results of Afek et al. [1] and Fekete, Lynch, Mansour, and Spinelli [6]. In [2], Attiya, Dolev, and Welch attain further results for the asynchronous model based on the minimum amount of information that must be maintained between incarnations of a connection in the presence of crashes. None of these papers examines the amount of time or the number of trips across the network required to reliably deliver messages.

The closest results to the work presented in this paper are the papers by Kleinberg, Attiya, and Lynch [10] and by Attiya and Rapport [3]. In [10] several impossibility results are obtained for connection management for various timing and failure assumptions. Additionally, upper and lower bounds are proved for trade-offs between message delivery and quiesce times for connection management protocols. A recent paper by Mavronicoles and Papadakis [13] extends the results of [10] by improving the time bounds in some of the trade-offs. Attiya and Rappoport [3] prove that in the absence of crashes, in an asynchronous setting where the client and server both have an infinite set of UID's and must quiesce, a three-way handshake is necessary to guarantee reliable message delivery. This result means that in such a setting it is not possible to always have transactions in one round across the network and still have reliable message delivery. They also show that if the server retains information between incarnations and there are no crashes, or if the MPL is known then fast transactions are possible.

The work presented in this paper differs form the results of [10] and [3] and other works in the literature in that we consider a more restricted problem. The problem is different because we do not require that transactions are fast at all times. We also treat packet loss differently. For example, in [10] the results for the model where packets may get lost are based the probability distribution of executions where packets are lost. In our work we take a simpler approach to packet loss. Our approach is to require fast transactions only if no packets are lost, and just eventual delivery otherwise. Our model of the client and server processes is also different than the models in [10] and [3] in that we allow either or both hosts to initiate a connection. In the models of [10] and [3] only the client can initiate a connection. While the problem we define is different from typical problem descriptions in the theoretical literature, it is not a contrived definition as it is based on the requirements for T/TCP.

1.3 Organization of paper

The rest of the paper is organized as follows. In Section 2 we present the formal model we use to describe clients, servers, channels and protocols. Section 2 also contains our formal definition of the problem. We state and prove the impossibility results in Section 3, and we make some concluding remarks in Section 4.

2 Formal models

In this section we present a brief overview of the formal model used in this paper. For full details of the

model the reader is referred to [18]. The basic underlying model is the *general timed automaton (GTA)* model of Lynch [12]. A GTA A consists of four components; a set, *states(A)*, of states; a nonempty set, *start(A)* $\subseteq$ *states(A)*, of start states; a set, *acts(A)*, of actions; and a set, *steps(A)* $\subseteq$ *states(A)* $\times$ *acts(A)* $\times$ *states(A)*, of steps. The set *acts(A)* can be partitioned into four disjoint sets of *input actions*, *output actions*, *internal actions*, and *time-passage actions*. Time-passage actions are of the form $\nu(t)$, $t \in R^+$, where R^+ is the set of positive reals. *Parallel composition* in this model uses a synchronization style where automata synchronize on their common actions and evolve independently on the others.

A *timed execution fragment* of a GTA A is a finite or infinite alternating sequence $\alpha = s_0, a_1, s_1, a_2, \ldots$, where, the s's are states of A and the a's are actions of A, and (s_i, a_{i+1}, s_{i+1}) is a step of A for every i. The sequence must begin with a state, and if it is finite must end with a state. For a timed execution fragment α, define *ltime(α)*, the *last time* of α, to be the supremum of the sum of all the time passage actions in α. A timed execution fragment α is defined to be *finite* if it is a finite sequence and *ltime(α)* is finite. It is defined to be *admissible* if $ltime(\alpha) = \infty$.

Let α be a timed execution fragment of a timed automaton that is the parallel composition of timed automata, and let A be one of the component timed automata. We define the *projection* of α on A to be the sequence obtained by projecting all states of the composed system onto those of A and removing actions not belonging to A. We use the notation $\alpha|A$ for the result of this operation. Informally, $\alpha|A$ is automaton A's view of timed execution fragment α. If $\alpha|A$ differs from $\alpha'|A$ only because of the splitting and combing of time-passage actions, then these are essentially the same views, and are said to be *time-passage* equivalent.

2.1 The clock GTA model

In the system we want to model, the client and server have access to local clocks, but are not able to use real time. However, a standard GTA automaton has access to real time. Thus, to get the "local clock" property, we use the *clock general timed automaton (CGTA)* model of De Prisco [15] which is a special case of the GTA model. A CGTA, A, is a GTA with a special variable $clock_A$ that has type $R^{\geq 0}$ and is the local time of that automaton. The local time may or may not be the same as real time. A CGTA A has the following three axioms: (1) $clock_A$ changes only with time passage actions, (2) $clock_A$ is monotonically non-decreasing, and (3) if $(s, \nu(t), s')$ is a step then $\forall\ t' > 0, (s, \nu(t'), s')$ is also a step. Since $clock_A$ is supposed to represent the local time of a process, real time should not affect the actions of the process in any manner. This property is captured by the third axiom. We also refer to this property as *real time independence*.

We introduce *clock functions* to specify the values that $clock_A$ takes on for a timed execution fragment for a given a CGTA A. These functions take real time as input and return values for the $clock_A$ variable of A. A clock function $cf : R^{\geq 0} \rightarrow R$ must be monotonically non-decreasing and it must be unbounded. Let A_{cf} be the CGTA we get by applying the clock function cf to A.

2.2 Liveness

The general timed automaton model is useful for proving safety properties and some liveness properties. However, for the impossibility results we prove, we need more general liveness properties than can be expressed by the GTA model. In particular, we want the automaton to not block time. To get this property we use a model defined by Segala et al. in [16]. We call it the *live GTA* model because its first component is a GTA. The second component of the model is a *liveness condition*. A *liveness condition* L for a GTA A is a subset of the timed execution fragments of A such that any finite timed execution of A has an extension in L. Thus, a live GTA is a pair (A, L) where A is a GTA and L is a liveness condition. For each live GTA we describe later in this work, the liveness condition is equal to the set of admissible timed executions of the GTA. Thus, we do not allow protocols that solve the problem by blocking time.

To get the liveness property we want and local clocks in the model, we combine the CGTA with the liveness property from the live GTA model to get the *live CGTA* model. For the proofs later in this work, we need a liveness property that relates admissible timed executions of live GTA to clock functions. Thus, our definition for a live CGTA is the following:

Definition 1 (Live CGTA)
A live CGTA is a pair (A, L) such that for every clock function cf, (A_{cf}, L) is a live GTA. □

2.3 Channels and client/server processes

We model the client and server as live CGTA and the channels as live GTA. When we describe a particular execution of the system, we apply clock functions to the CGTA to get values for *clock* variables. The parallel composition of the client, channels, and server forms the system. Because parallel composition of live GTA is closed [16], the resulting composed system is also a live GTA.

The communication channels have the following properties: packets placed in a channel are delivered in FIFO order; packets are not duplicated; and if infinitely many copies of a packet p get sent on a channel, then infinitely many copies of p are received. The last property is the strong loss limitation (SLL) property of channels defined by Lynch in [12]. If the channels have a maximum packet lifetime, μ, we refer to them as μ-SLL-FIFO channels, otherwise we refer to them as SLL-FIFO channels. The channels we describe here are more reliable than the typical representation of unreliable network channels in that they are FIFO and do not duplicate packets. Thus, our results are technically stronger than results that require non-FIFO channels that may duplicate messages.

The client and server processes are modeled by live CGTA (C, L) and (S, L') respectively, where L is the set of admissible timed executions of C and L' is the set of admissible timed executions of S. The client and server each has a set of UID's. We model the UID's by augmenting the state of C and the state of S with the addi-

tional components I_C and I_S respectively. These components are infinite sets of abstract identifiers. Since only the internal component of the states of a process is reset after it crashes, the sets of UID's are not affected by crashes. The UID's can be copied and included in packets. However, each process can only perform the following operations on its own set: *generate()* which nondeterministically returns a new UID from the host's set of UID's and removes that id from the set, and *same(x,y)* which returns true if and only if $x = y$, where x and y are UID's, and false otherwise.

In an admissible timed execution where clock values are determined by clock functions, after a crash there is an eventual recovery that returns the crashed host to an initial state. We assume that local clocks are not affected by crashes. Since we are concerned only with the delivery of messages by the server, and for our proofs we need to allow crashes only at the server, we use the following user interface actions: *send(m)* is the input action at the client to send a message m, *deliver(m)* is the output action at the server that delivers m, *crash* is the input action that signals a crash at the server, and *recover* is the output action that indicates the server has recovered from a crash. Additionally, both client and server can place packets on and receive packets from the channels.

2.4 Formal definition of the problem

We now present a formal abstract definition of the problem that T/TCP was designed to solve. We call it the *conditionally-fast reliable message delivery* problem. Recall that T/TCP is only expected to have fast transactions if the client and server have sufficient state information. Otherwise, the protocol is just required to deliver messages reliably. In T/TCP, if the server successfully delivers a message, and there has not been a crash since the delivery of that message, then there is enough state information to perform subsequent transactions in one round trip. Thus, in our formal definition we use the successful delivery of a message as the user visible indication of sufficient state information for fast transactions. Let d be the maximum packet delay on the channels. We define the delivery of a message to be fast if the server delivers the message in time strictly less than $2d$ from the time the client receives the input to send the message. For fast transactions, a delivery time of less than or equal to d is actually needed. However, for our proof we only need fast to be less than $2d$, which makes our results technically stronger.

Definition 2 (The Conditionally-fast reliable message delivery problem)
Reliable delivery. Messages are always delivered at most once and in the right order. That is, for every execution there exists a function *cause* that maps *deliver* actions to preceding *send* actions such that for every *deliver* action π, π and *cause*(π) have the same message argument; *cause* is one-to-on; and for any two *deliver* events π_1 and π_2, if π_1 precedes π_2, then *cause*(π_1) precedes *cause*(π_2).
Eventual delivery. In an admissible timed execution where clock values are determined by clock functions the following two conditions hold. If there are no crashes then all messages are delivered, and if there are finitely many crashes, messages sent after the last *crash* action and the subsequent *recover* action are eventually delivered.
Conditionally-fast delivery. For any admissible timed execution in which there is a *deliver(m')* action (for any message m') and the client subsequently receives a *send(m)* input, if the following conditions hold:

1. the clocks of the client and server always run at the rate of real time;
2. both sides are recovered at the time of the *deliver(m')* action, and there are no *crash* or *recover* events after the *send(m)* input;
3. any packet sent by the client or server after the client receives the *send(m)* action from the user takes time at most d to arrive at its destination;

then the server performs *deliver(m)* in time strictly less than $2d$ after the client receives the *send(m)* input. □

3 Impossibility results

3.1 Maximum packet lifetime exists

The first result is for the case where there is a maximum packet lifetime, μ. We get an impossibility result for this situation if the clocks of the client an the server may run at arbitrary rates.

Theorem 1 *No system consisting of μ-SLL-FIFO channels and client/server processes can solve the conditionally-fast reliable message delivery problem.*

Proof: Our proof strategy is to construct executions that behave as required by the problem definition, and then show that we can construct another execution that is a sort of combination of the previous executions, but where the new execution has incorrect behavior.

We start by assuming we have a protocol that solves the conditionally-fast reliable message delivery problem, and show that this assumption leads to a contradiction. Throughout the proof we mention the real time at which different events occur even though the client and server do not have access to real time. The local clocks are $clock_C$ and $clock_S$ for the client and server respectively. In an execution, the values for these clocks are determined by the clock functions we describe. In all the executions we construct $clock_C$ is equal to real time; that is, the clock function of the client is the identity function for all executions.

The first execution we construct, α_1, is shown in Figure 1. It is an admissible timed execution as are all the executions we construct for this proof. In this execution, $clock_S$ is also equal to real time. Since both the client and server are recovered at time 0, α_1 is an admissible timed execution, and there are no *crash* or *recover* actions after the *send(m')* input, the eventual delivery property results in the server action *deliver(m')*. Let $clock_S = p$, which is also real time p, be the time of this action. Only the packets required for the delivery of m' are received up to time p. Execution α_1 continues as follows. At real time $p + 2\epsilon$, where ϵ is an arbitrary constant greater than 0, the client receives a *send(m)* input and all packets sent by both the client and server

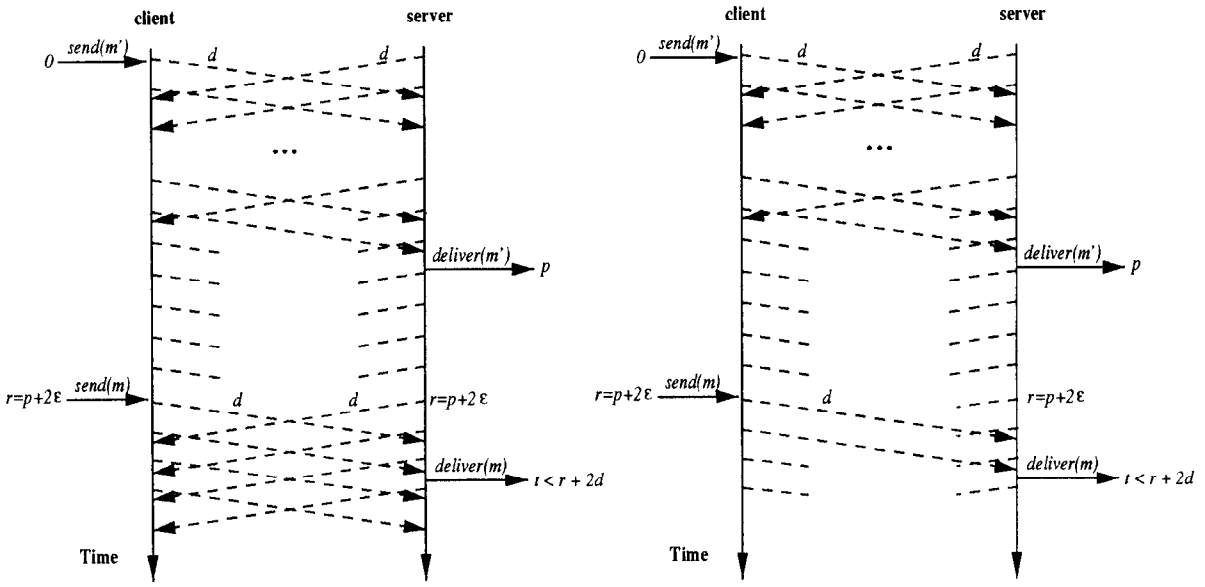

Figure 1: *Execution α_1 is on the left and execution α_2 is on the right. The numbers outside the time lines represent real time, the dashed lines represent packets, and the "$\cdots$" between packets represents a finite number (whatever the protocol needs) of packets in both directions. The numbers on the dashed lines represent the time it takes the packets to traverse the channel. Execution α_1 is the same as α_2 except that additional packets are dropped from the channels in α_2.*

after this input action take time d to arrive. Because the value $p+2\epsilon$ appears in several subsequent executions we construct, to simplify the notation we let $r = p + 2\epsilon$. Since we assume the protocol satisfies the conditionally-fast delivery property, at some real time $t < r + 2d$ the server delivers the message m. For execution α_1, let U_c^1 and U_s^1 be the set of UID's used by the client and the server respectively.

Now consider the execution α_2, shown in Figure 1. This execution is exactly the same as execution α_1 except that all the packets sent by the server after the *send(m)* event get dropped by the channel, and all packets sent by the client at or after time $r + d$ also get dropped from the channel. However, from time 0 up to and including time t, $\alpha_2|S$ is time passage equivalent to $\alpha_1|S$. Thus, at time t in execution α_2 the server can deliver m. The bound of less than $2d$ on delivery time is important here because it forces the server to deliver the message even though the client has not received any packets from the server since the *send(m)* event.

The next execution α_3 is shown in Figure 2. Let U_c^3 be the set of UID's used by the client, and let U_s^3 be the set used by the server. For parts of this execution $clock_S$ runs at the rate of real time and for other parts it runs faster than the rate of real time. We define the clock function for the server by giving the rate of $clock_S$ relative to real time for different real time intervals. For the real time interval $[0, p]$, $clock_S$ runs at the rate of real time and looks like execution α_2, except for the fact that the UID's used may be different. However, because the only two operations that can be performed on UID's are *generate()* and *same*, if in execution α_2 a host receives a packet with UID u and performs the operation *same(u,v)* for some UID v, and if at the same time in execution α_3, the same host receives a packet with UID x and performs the operation *same(x,y)* for some UID y, then *same(u,v)* = *same(x,y)*. Thus, the fact that U_c^3 and U_s^3 are used in execution α_3 does not affect the behavior of α_3 relative to α_2 up to time p. Therefore, in execution α_3, at time p the server can perform the *deliver(m')* action.

Throughout the rest of this proof we compare executions where packets are sent and received at the same local clock time, but where the packets may have different sets of UID's. The argument just presented can be applied to all these comparisons to show that the use of different sets of UID's cannot cause the client or server to behave differently in our model under these circumstances.

After the *deliver(m')* action and up to real time $p+\epsilon$, that is, for the interval $(p, p + \epsilon]$, $clock_S$ runs at $(2\epsilon +$

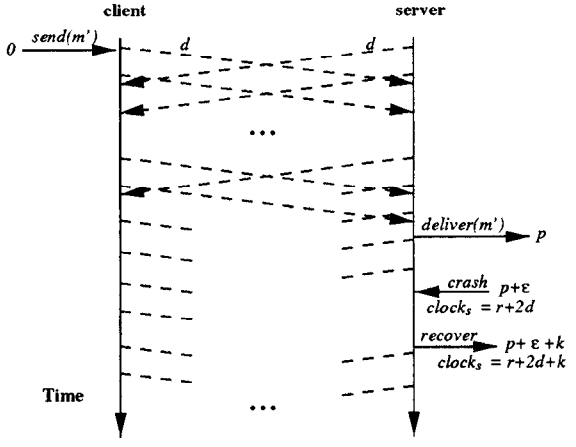

Figure 2: *Execution α_3. Values of* clock$_S$ *are shown under the corresponding real time values.*

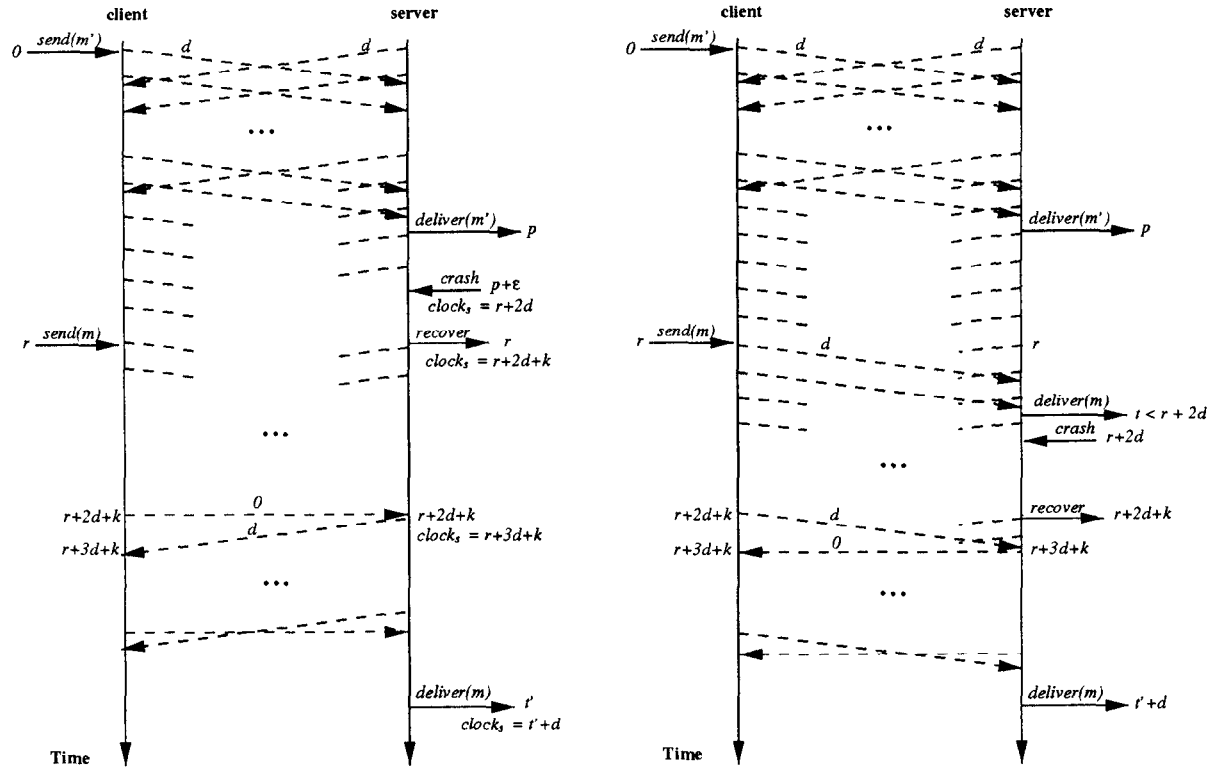

Figure 3: *Execution α_4 is on the left and execution α_5 is on the right. Execution α_4 is an extension of α_3 to include some additional sending and receiving of packets. It also includes an additional send action and the subsequent delivery. Execution α_5 demonstrates how the reliable delivery property can be violated. It combines parts of executions α_2 and α_4.*

$2d)/\epsilon$ times the rate of real time, and from time $p+\epsilon$ through the rest of the execution, that is, the interval $(p+\epsilon, \infty)$, $clock_S$ runs at the rate of real time again. Now let the server receive a *crash* input at real time $p+\epsilon$. Because of the rate of $clock_S$ for the interval $(p, p+\epsilon]$, at real time $p+\epsilon$, $clock_S = p + 2\epsilon + 2d = r + 2d$. Since α_3 is an admissible timed execution and $clock_C$ and $clock_S$ are determined by clock functions, the server eventually recovers. The time of recovery is determined by the protocol, but it must happen after the *crash* event. Let k be the $clock_S$ time between crash and recovery. Since $clock_S$ is now running at the rate of real time, k is also the difference in real time between the *crash* input and the *recover* output. Thus, the recovery happens when $clock_S = r + 2d + k$, which is real time $p + \epsilon + k$.

The next execution, α_4, shown in Figure 3, starts out like execution α_3 except the client and server use the sets of UID's U_c^4 and U_s^4 respectively, and the clock function of the server is different from the clock function in execution α_3. The clock functions are the same for the real time interval $[0, p+\epsilon)$. However, for the real time interval $(p+\epsilon, p+2\epsilon]$, $clock_S$ runs at k/ϵ times the rate of real time. Therefore, in execution α_4 when $clock_S = r + 2d + k$ it is real time r. Because of the real time independence property, we know that when $clock_S = r + 2d + k$ in this execution, the server can perform the *recover* action. After the *recover* action through real time $r+2d+k$, that is, the real time interval $(r, r+2d+k]$, $clock_S$ runs at $d/(2d+k)$ times the rate of real time. Therefore, at real time $r + 2d + k$, $clock_S = r + 3d + k$. After that time through the rest of the execution, that is, the real time interval $(r + 2d + k, \infty)$, $clock_S$ runs at the rate of real time. After the *recover* event at the server, the client gets the *send(m)* input at real time r at which time $clock_S = r + 2d + k$. However, all of the packets that both the client and server send from real time r up to, but not including real time $r + 2d + k$ ($clock_S = r + 3d + k$) are dropped from the channels. All packets sent by the client and server starting at real time $r+2d+k$ do not get dropped from the channels and take time 0 and d to arrive respectively. Execution α_4 is an admissible timed execution, $clock_C$ and $clock_S$ are determined by clock functions, and m is sent after the last *crash* and *recover* actions. Therefore, since the protocol satisfies the *eventual delivery*, property the server must eventually perform the *deliver(m)* action. Let real time t' and $clock_S = t' + d$ be the time of this event.

Finally, we construct an execution α_5 where the server delivers the same message twice. This execution is shown in Figure 3. In the execution, $clock_S$ runs at the rate of real time for the whole execution. The client uses the set of UID's U_c^5 and the server uses the set U_s^5. On the client side, except for the use of a different set of UID's, execution α_5 is exactly the same as execution α_4, so *send(m')* happens at time 0, and *send(m)* happens at real time r. However, in execution α_5 the packets the client sends after the *send(m)* input and before time $r+d$ are not dropped from the channel. On the server side, except for the use of a different set of UID's, from time 0 to time t execution α_5 looks the

same as execution α_2. That is, modulo the UID's, for the time interval $[0,\ t]$ execution $\alpha_5|S$ is time passage equivalent to $\alpha_2|S$. Therefore, since in execution α_2 the server performs *deliver(m')* at time p and *deliver(m)* at time t, in execution α_5 it can do likewise.

For the rest of α_5, the packets the client sends at or after time $r+d$ until, but not including time $r+2d+k$, are dropped from the channel, and on the server side at time $r+2d$ a *crash* input occurs. For the real time interval $[r+2d, r+2d+k]$, $\alpha_5|S$ is time passage equivalent to $\alpha_4|S$. Therefore, because of the real time independence property, at $clock_S = r+2d+k$, the server can perform the *recover* output action. Any packet sent by the server after the *recover* event up to, but not including time $r+3d+k$ is dropped from the channel. The packets that the client sends starting at time $r+2d+k$ take time d to arrive at the server, and the packets that the server sends starting at time $r+3d+k$ take time 0 to arrive at the client. Except for the fact that packets sent and received may have different UID's, in the $clock_S$ interval $[r+2d,\ t'+d]$ in execution α_5 the server receives exactly the same inputs as in the same $clock_S$ interval in execution α_4. Since the *recover* action returns the server to an initial state where it does not remember any previous actions in both executions, modulo packet UID's, $\alpha_5|S$ is time passage equivalent to $\alpha_4|S$ for the $clock_S$ intervals $[r+2d,\ t'+d]$. Because of the real time independence property of the server, we know that at $clock_S = t'+d$ the server can perform the *deliver(m)* action. Since m was already delivered, we have duplicate delivery which contradicts our assumption that the protocol delivers messages reliably. ■

3.2 No maximum packet lifetime

If there is no MPL, then we get the impossibility result with a more realistic clock model. Here we assume that the local clocks of the client and server always run at the rate of real time, but their values may be shifted by an unknown amount. We call these shifted clocks. If the values of $clock_C$ and $clock_S$ in an execution are determined by the clock functions cf_c and cf_s respectively, then our assumption can be expressed by saying that for all values t_1 and t_2 of real time, $cf_c(t_1) - cf_c(t_2) = cf_s(t_1) - cf_s(t_2) = t_1 - t_2$.

Theorem 2 *No system consisting of SLL-FIFO channels and client/server processes with shifted clocks can solve the conditionally-fast reliable message delivery problem.*

Proof: The proof is very similar to the proof of Theorem 1. First we construction an execution were fast deliver occurs. This execution, β_1 is similar to execution α_2 and is shown in Figure 4. Execution β_1 is an admissible timed execution as are all the executions constructed in this proof. For this execution both the clock of the client and the clock of the server run at the rate of real time. Since both the client and server are recovered at time t_0 when the client receives a *send(m')* input and there are no *crash* or *recover* actions after this input, the eventual delivery property results in server action *deliver(m')*. Let this be real time p, which for this execution also means $clock_S = p$. In this execution the packets sent by the client after the input at time t_0 up

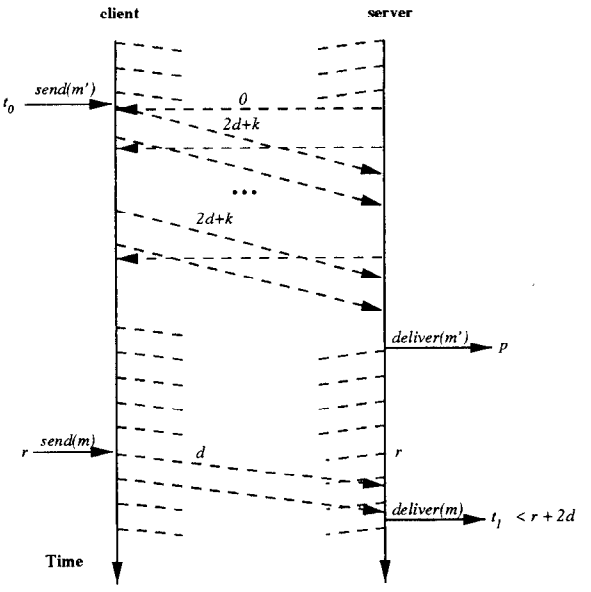

Figure 4: *Execution β_1. This execution is similar to execution α_2.*

to last packet required for the delivery of m' take time $2d+k$ to arrive, and the packets sent by the server takes time 0 to arrive. At some time r where $r > p+2d+k$, the client receives a *send(m)* input and all packets it sends from that time, up to, but not including time $r+d$ take time d. All packets sent at or after time $r+d$ by the client get dropped, as do all the packets sent by the server after the *send(m)* input. However, as we demonstrated in the proof of Theorem 1, because the protocol must satisfy the conditionally-fast delivery property, at some real time $t_1 < r+2d$ the server delivers the message m.

The next execution we construct, β_2 is similar to execution execution α_4 and is shown in Figure 5. In this execution the clock of the server is shifted forward by $2d+k$. That is, at any real time t, $clock_S = t+2d+k$. In execution β_2, the client still gets the *send(m')* input at real time t_0. However, this time the packet it sends take time 0 to arrive at the server. At $clock_S = t_0$, which is real time $t_0 - 2d - k$, the server starts sending packets that take time $2d+k$ to arrive at the client. Because of the shift in the clock of the server, and the differences in times the packets take to traverse the channels, for the $clock_S$ interval between t_0 and p, modulo packet UID's, execution β_2 looks the same as execution β_1 to the server. That is, for the $clock_S$ interval $[t_0, p]$, $\beta_2|S$ is time passage equivalent to $\beta_1|S$. Therefore, because of the real time independence property, we know the server can deliver m' at $clock_S = p$ in execution β_2. After the *deliver(m')* event, the server receives a *crash* input at $clock_S = r+2d$ which is real time $r-k$. At $clock_S = r+2d+k$ the server recovers. After the *recover* event at the server, the client gets the *send(m)* input at real time r at which time $clock_S = r+2d+k$. All of the packets that both the client and server send from real time r up to, but not including real time $r+2d+k$ ($clock_S = r+4d+2k$) are dropped from the channels. All packets sent by the client and server starting at real

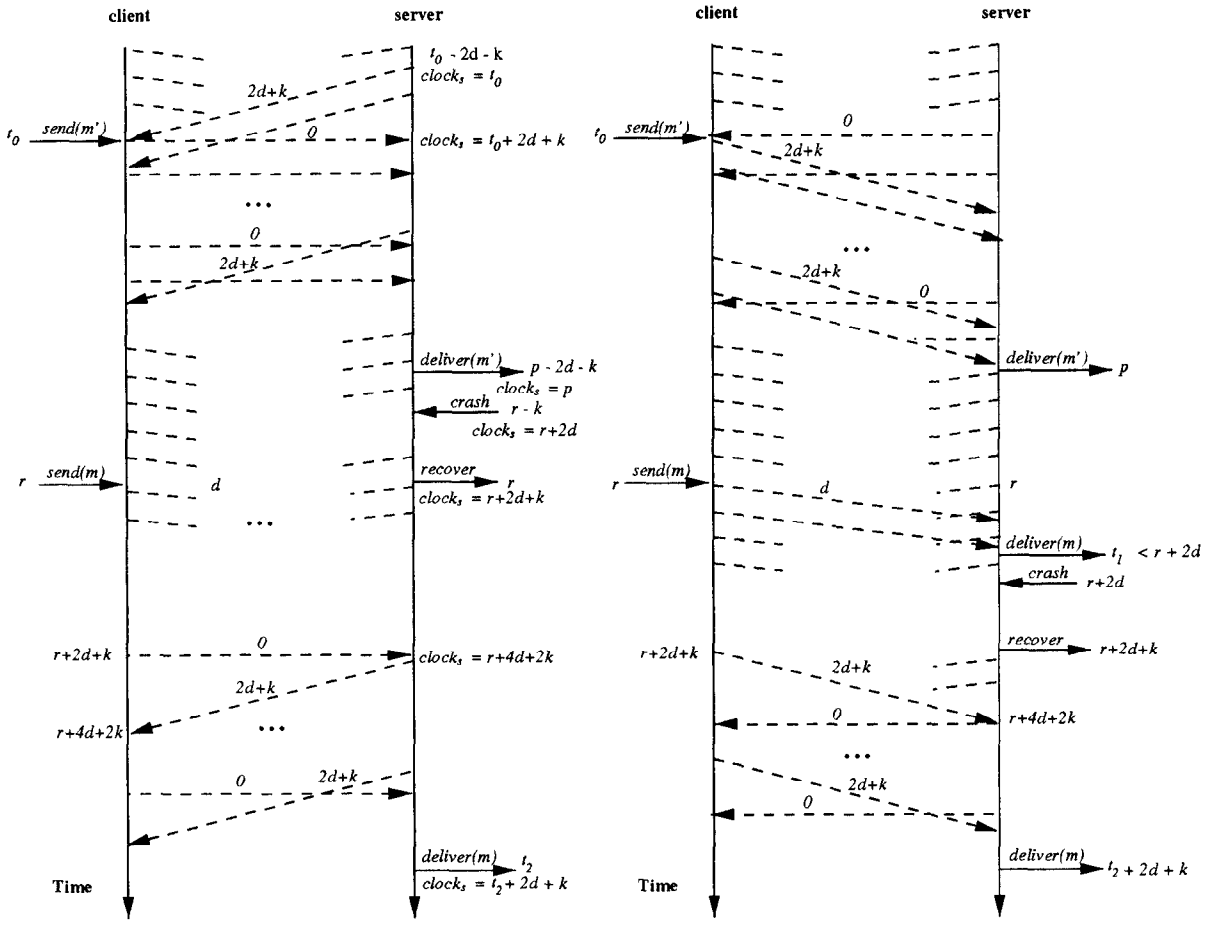

Figure 5: *Execution β_2 is on the left and execution β_3 is on the right. These executions are similar to executions α_4 and α_5 respectively.*

time $r + 2d + k$ do not get dropped from the channels and take time 0 and $2d + k$ to arrive respectively. Execution β_2 is an admissible timed execution, $clock_C$ and $clock_S$ are determined by clock functions, and m is sent after the last *crash* and *recover* actions. Therefore, since the protocol satisfies the *eventual delivery*, property the server must eventually perform the *deliver(m)* action. Let real time t_2 and $clock_S = t_2 + 2d + k$ be the time of this event.

The final execution we construct for this proof is β_3. It is similar to execution α_5 and is shown in Figure 5. Here the clock of the server is again the same as real time. On the client side, except for the possible use of different UID's, execution β_3 is exactly the same as execution β_2. Thus, *send(m')* happens at real time t_0 and *send(m)* happens at real time r. However, in the this execution, for the real time interval $[t_0, t_1 - d)$, the packets sent by the client have same delivery times as packets sent by the client in execution β_1. Similarly, on the server side, modulo UID's, for the interval $[t_0, t_1]$, $\beta_3|S$ is time passage equivalent to $\beta_1|S$. Therefore, at time p and time t_1 the server can deliver m' and m respectively.

For the rest of β_3, the packets the client sends at or after time $r + d$ until, but not including time $r + 2d + k$, are dropped from the channel, and on the server side at time $r + 2d$ a *crash* input occurs. For the real time interval $[r + 2d, r + 2d + k]$, $\beta_3|S$ is time passage equivalent to $\beta_2|S$. Therefore, because of the real time independence property, at $clock_S = r+2d+k$, the server can perform the *recover* output action. Any packet sent by the server after the *recover* event up to, but not including time $r + 4d + 2k$ is dropped from the channel. The packets that the client sends starting at time $r + 2d + k$ take time $2d + k$ to arrive at the server, and the packets that the server sends starting at time $r + 4d + 2k$ take time 0 to arrive at the client. Except for the fact that packets sent and received may have different UID's, in the $clock_S$ interval $[r + 2d,\ t_2 + 2d + k]$ in execution β_3 the server receives exactly the same inputs as in the same $clock_S$ interval in execution β_2. Since the *recover* action returns the server to an initial state where it does not remember any previous actions in both executions, modulo packet UID's, $\beta_3|S$ is time passage equivalent to $\beta_2|S$ for the $clock_S$ intervals $[r + 2d,\ t_2 + 2d+k]$. Because of the real time independence property of the server, we know that at $clock_S = t_2 + 2d + k$ the server can perform the *deliver(m)* action. Since m was already delivered, we have duplicate delivery which contradicts our assumption that the protocol delivers

messages reliably.

In the executions in the proof, the delay on some packets is $2d + k$. Since k may be some arbitrary value set by the server, the proof requires that there is no MPL. ■

4 Conclusion

There has been significant theoretical results [1, 6, 2, 3, 10, 13] on the limitations of connection management and reliable message delivery protocols under various timing and failure assumptions. Our work adds a new dimension to traditionally studied problems by adding conditional requirements. In our work we formally define what we call the conditionally-fast reliable message delivery problem. The definition is based on requirements for T/TCP which is a TCP/IP transport level protocol designed to support both reliable data streaming and fast transactions. When there is a maximum packet lifetime, we proved that it is impossible for any protocol to solve this problem if the local clocks may sometimes run at arbitrary rates. If there is no MPL, we prove that if the clocks run at the rate of real time, but may be shifted from real time by some arbitrary amount, then again it is impossible to solve the problem. The problem definition and the proofs are presented in a carefully developed formal model, which is a novel combination of a model with liveness properties and a model that allows local clocks.

Acknowledgments

Nancy Lynch was a source of very useful comments and discussions for this work. Her comments were particularly insightful in discussions of the formal model.

References

[1] AFEK, Y., ATTIYA, H., FEKETE, A., FISCHER, M., LYNCH, N., MANSOUR, Y., WANG, D.-W., AND ZUCK, L. Reliable communication over unreliable channels. *Journal of the ACM 41*, 6 (November 1994), 1267–1297.

[2] ATTIYA, H., DOLEV, S., AND WELCH, J. Connection management without retaining information. Technical Report LPCR 9316, Laboratory for Parallel Computing Research, Dept. of Computer Science, The Technion, June 1993.

[3] ATTIYA, H., AND RAPPOPORT, R. The level of handshake required for managing a connection. In *The 8th International Workshop on Distributed Algorithms* (September/October 1994), G. Tel and P. Vitanyi, Eds., no. 857 in Lecture notes in Computer Science, Springer-Verlag, pp. 179–193.

[4] BRADEN, R. Extending TCP for transactions – concepts. Internet RFC-1379, November 1992.

[5] BRADEN, R., AND CLARK, D. Transport protocols for transactions and streaming. Unpublished manuscript, March 1993.

[6] FEKETE, A., LYNCH, N., MANSOUR, Y., AND SPINELLI, J. The impossibility of implementing reliable communication in the face of crashes. *Journal of the ACM 40*, 5 (November 1993), 1087–1107.

[7] GAWLICK, R., SEGALA, R., SØGAARD-ANDERSEN, J., AND LYNCH, N. Liveness in timed and untimed systems. Technical Report MIT/LCS/TR-587, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, 02139, December 1993.

[8] GAWLICK, R., SEGALA, R., SØGAARD-ANDERSEN, J., AND LYNCH, N. Liveness in timed and untimed systems. In *Automata, Languages and Programming* (21st International Colloquium, ICALP'94, Jerusalem, Israel, July 1994) (1994), S. Abiteboul and E. Shamir, Eds., vol. 820 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 166–177. Full version in [7]. Also, submitted for publication.

[9] INTERNATIONAL STANDARDS ORGANIZATION. *Connection Oriented Transport Protocol Specification*, 1986. International Standard 8073, ISO.

[10] KLEINBERG, J., ATTIYA, H., AND LYNCH, N. Trade-offs between message delivery and quiesce times in connection management protocols. In *Proceedings of 3rd Israel Symposium on Theory of Computing and Systems* (Tel-Aviv, Israel, January 1995), pp. 258–267.

[11] LISKOV, B., SHRIRA, L., AND WROCLAWSKI, J. Efficient at-most-once messages based on synchronized clocks. *ACM Transactions on Computer Systems 9*, 2 (May 1991).

[12] LYNCH, N. *Distributed Algorithms*. Morgan Kaufmann Publishers, Inc, 1996.

[13] MAVRONICOLAS, M., AND PAPADAKIS, N. Trade-off results for connection management. In *Proceedings FCT'97* (September 1997), pp. 340–351.

[14] POSTEL, J. Transmission Control Protocol - DARPA Internet Program Specification (Internet Standard STC-007). Internet RFC-793, September 1981.

[15] PRISCO, R. D. Revisiting the Paxos algorithm. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139, 1997.

[16] SEGALA, R., GAWLICK, R., SØGAARD-ANDERSEN, J., AND LYNCH, N. Liveness in timed and untimed systems, 1997. Submitted for publication. An earlier version appears in [7] and a shortened version appears in [8].

[17] SHANKAR, A. U., AND LEE, D. Minimum-latency transport protocols with modulo-n incarnation. *IEEE/ACM Transactions on Networking 3*, 3 (June 1995), 255–268.

[18] SMITH, M. *Formal Verification of TCP and T/TCP*. PhD thesis, M.I.T., 1997.

[19] WATSON, R. W. The delta-t transport protocol: Features and experience. In *IEEE 14th Conference on Local Computer Networks* (October 1989), pp. 399–407.