SOLVING THE RMTP WITH AN UNKNOWN BOUND ON REORDERING USING BOUNDED COUNTERS

An Undergraduate Research Scholars Thesis

by

GRANT KIRCHHOFER

Submitted to the Undergraduate Research Scholars program at
Texas A&M University
in partial fulfillment of the requirements for the designation as an

UNDERGRADUATE RESEARCH SCHOLAR

Approved by Research Advisor: Dr. Jennifer Welch

May 2018

Major: Computer Science and Engineering

TABLE OF CONTENTS

ABSTRACT

Solving the RMTP with an Unknown Bound on Reordering using Bounded Counters

Grant Kirchhofer
Department of Computer Science and Engineering
Texas A&M University

Research Advisor: Dr. Jennifer Welch
Department of Computer Science and Engineering
Texas A&M University

This research analyzes the reliable message transmission problem, or RMTP, with a different set of constraints than has been previously studied. The RMTP describes the task of simulating a reliable computer communication channel on top of an unreliable one. The unreliable channel can exhibit undesirable behavior, including message loss, duplication, and reordering. The reliable channel exhibits none of these. Prior research has proposed an algorithm that solves the RMTP using bounded message counters when the channel exhibits duplication and bounded reordering, where the bound on reordering is known. This paper studies a variation of that configuration with an unknown bound on reordering. Using well documented C++ code, data collected from experimental executions of that code, and formal logical and mathematical proofs, we show that for several classes of algorithms, there is no possible algorithm that solves the RMTP where the bound on reordering is unknown. We also develop an algorithm that can, with enough input, solve the RMTP when the reordering bound is unknown but is within a known range.

1. INTRODUCTION AND LITERATURE REVIEW

1.1 Background

The reliable message transmission problem, or RMTP, is the problem of implementing a reliable layer of communication on top of an unreliable layer, where an unreliable channel displays undesirable behavior. The RMTP is important because no communication channel, especially a wireless one, is absolutely reliable. Safeguards against inherent unreliability must be implemented in order to simulate reliability as closely as possible.

An unreliable communication channel exhibits certain types of undesirable behavior. The three main types of undesirable behavior are loss, duplication, and reordering. Loss is when a message is sent through the channel, but is not received on the other end. Duplication is when a message is received multiple times for only one message sent. Reordering is when a message is sent before but received after another message. However, it is assumed that the contents of the messages are preserved through the channel. This research will not study message data corruption. In the context of distributed algorithms and communication channels, asynchronous means that the time between the message being sent and received is unknown and unbounded.

1.2 Prior Research

Older research in this area [1] has found that in purely asynchronous systems, assigning unbounded sequence numbers to messages allows for the tolerance of unbounded loss, duplication, and reordering. However, unbounded sequence numbers require unbounded space to store them, which is not ideal in real systems. The Alternating Bit Protocol [2] assigns a single identifier bit to each message, and it solves the RMTP when only loss or only duplication is considered. For both loss and duplication, additional header

information is required. When loss and reordering with duplication is considered, the sender must send an unbounded number of copies of the same message until the receiver has notified the sender of receipt [3]. When duplication and reordering are considered, no bounded solution is possible [4].

More recent work, in particular that of Ortiz-Lopez and Welch [5], has studied partially synchronous systems, where the undesirable behavior is bounded. In other words, a message can only be duplicated so many times, only so many consecutive messages can be lost, and two messages sent one after another can only have so many messages received between them. The KJ Protocol in [5] solves the RMTP when considering unbounded duplication and bounded reordering. It does this using bounded counters; a counter between zero and an upper bound is attached to each message. The Extended KJ Protocol in [5] is similar but also considers bounded loss and handles it by sending n copies of each message, where n is one more than the bound on the number of consecutive messages that can be lost.

My research will build upon the study of partially synchronous systems. Specifically, I will develop an algorithm that addresses the RMTP when loss is bounded, duplication is unbounded, and reordering is bounded with an unknown bound, as opposed to a known bound. I will see whether it is possible to solve this scenario using bounded counters.

1.3 Overview of Contribution

I have developed several theorems with formal proofs that show that under various circumstances, no algorithm can solve the RMTP. For convenience, I have named certain classes of Sender-Receiver algorithms; the Tag Class, and the Backchannel Class.

The first theorem states that when the Sender is the KJ Sender, as in [5], no algorithm can solve the RMTP with an unknown reordering bound. The second theorem states when the Sender is a Tag Class Sender, no algorithm can solve the RMTP with an unknown

reordering bound. The third theorem states that the RMTP with an unknown reordering bound cannot be solved using the Adjustment Interface, a high-level description of a way to use a Backchannel Class Sender-Receiver.

I have also developed an algorithm that solves the RMTP with an unknown reordering bound, but only when the bound is within a known range. This algorithm assumes that it is preferable to use less memory per message, and so tries to minimize the amount of extra data attached to each message.

I wrote a C++ program in Visual Studio 2015 that implements this algorithm. This implementation can be found at https://github.com/cyber5/AdjustmentInterfaceRMTP.

1.4 Structure of Thesis

The rest of this thesis is organized into sections as follows. Section 2 reviews relevant definitions and terms that will be used in the rest of the thesis, as well as some useful properties. Section 3 contains multiple theorems and impossibility proofs concerning the RMTP with an unknown reordering bound when using various classes of Sender algorithms. Section 4 contains a description and pseudocode of an algorithm that can, with enough input, solve the RMTP when the reordering bound is unknown but falls within a certain known range. Section 5 contains a conclusion and thoughts about what further research may yield.

2. INTRODUCION TO KEY CONCEPTS

2.1 Definition of Terms

We define a Sender that contains an algorithm whose attributes will be specified before each theorem. We define a Receiver that contains no specified algorithm. We have an unreliable channel that transmits low-level messages from the Sender to the Receiver. The channel is described in greater detail later.

We define an algorithm to be a combination of a Sender and a Receiver, both modeled as state automata. An algorithm takes a sequence of high-level messages as input; the Sender produces low-level messages and sends them to the Receiver across the unreliable channel; the algorithm's output is the sequence of high-level messages produced by the Receiver. Typically, the purpose of an algorithm is to simulate a reliable channel using the Sender and Receiver. This means correcting the unreliable behavior of the channel. We define a correct algorithm to be one that produces an output that matches the input exactly.

We define four types of events that can occur:

$SEND(M)$ — the Sender takes the next high-level message M from the input sequence

$send(m)$ — the Sender sends low-level message m across the channel

$receive(m)$ — the Receiver receives low-level message m from the channel

$RECEIVE(M)$ — the Receiver produces high-level message M and appends it to the output sequence

The Sender in the KJ Protocol, the "KJ Sender," is simple and straightforward. In the KJ Protocol, $SEND(M)$ creates a low-level message m from high-level message M, assigns it a bounded counter, and inserts it into a FIFO queue. It also increments the counter, which wraps around upon reaching the bound. The event $send(m)$ removes the

message at the head of the queue and sends it across the channel.

We define low-level messages with bounded counters and unbounded counters, the same as in the KJ Protocol. The unbounded counters are only read during the proofs, not during the execution of any algorithm.

We define $M_0M_1...M_{n-1}$ to be the finite sequence of n high-level messages that is input to the algorithm. It is also possible for the input sequence to be infinite $(M_0M_1...)$. We define m_i to be a low-level message that contains high-level message M_i. We define m_i^x to be m_i appended with the bounded counter x.

Event E is said to be enabled at state S if S can transition to another state as a result of E. We define an execution to be a sequence of alternating states and events, $S_0E_1S_1E_2S_2...E_nS_n$ beginning with the initial state of the automata and ending with the final state of the automata. An execution can also be infinite, provided that the input sequence is infinite. The initial state, S_0, describes the state of the Sender and Receiver before the algorithm begins and before input is received. For every subsequent $S_{i-1}E_iS_i$ in the execution, it must be the case that E_i is the event that transforms the automata from the state S_{i-1} to the state S_i. The final state, S_n, describes the state of the Sender and Receiver after the entire output sequence has been produced and the algorithm is finished. We define a schedule of an algorithm to be a sequence of events that occur over the course of the algorithm's execution; in other words, an execution without the states. We only consider fair executions, meaning that an event cannot be continually enabled but never occur.

We define the r-schedule to be the subsequence of a schedule made up only of the receive events. We define r_m to be the sequence of low-level messages in the r-schedule. We define $r_m{}'$ to be any prefix of the full sequence r_m.

All correct executions exhibit both safety (meaning that in a prefix of the execution, the sequence of high-level messages in the $RECEIVE$ events is a prefix of the sequence of

high-level messages in the $SEND$ events) as well as liveness (meaning that the number of $RECEIVE$ events equals the number of $SEND$ events). Additionally, in correct executions, $RECEIVE(M)$ and $receive(m)$ cannot appear before $SEND(M)$ and $send(m)$, respectively.

We say that m_i is enabled if $receive(m_i)$ is enabled. We define the enabled-set, E, to be the set of low-level messages that are enabled with respect to a certain execution prefix, or a given $r_m{}'$. E is static for a given $r_m{}'$ and dynamic across the entire sequence r_m.

We define the trivially-enabled-set, T, to be the subset of E containing only messages m for which $receive(m)$ would not remove any messages from E (i.e., no messages would become disabled). Note that the enabled-set is not concerned with which send events have occurred; we assume that any $receive(m)$ is preceded by $send(m)$ in a schedule.

The channel exhibits unbounded but finite duplication (meaning that there is no limit to the number of times the message will duplicate, but the duplication is guaranteed to stop eventually), no loss, no corruption, and bounded reordering with a bound of δ. The formal definition of the reordering bound δ is as follows: suppose we have low-level messages m_i and m_j, where $j \geq i + \delta$. Then the first occurrence of $receive(m_j)$ in the schedule must occur after the last occurrence of $receive(m_i)$. It follows that $receive(m_i)$ is enabled for a given $r_m{}'$ if every message $m_{i-\delta}$, $m_{i-\delta-1}$, ... m_{i-1} is in $r_m{}'$, and no message $m_{i+\delta}$, $m_{i+\delta+1}$, ... m_n is in $r_m{}'$.

2.2 Useful Properties of the Enabled-Set

There are some properties about E and T that can be beneficial for understanding them. While they are not used in this paper's theorems, they formed a basis for developing the algorithm.

Property 1. *If a message m has appeared in $r_m{}'$ and $m \in E$, then $m \in T$.*

Proof. By the definition of enabled, whether a message is enabled is solely dependent on

which messages appear in $r_m{}'$. Since m has already appeared in $r_m{}'$, $receive(m)$ would not change which messages appear in $r_m{}'$. Therefore no messages would become enabled or disabled. Since no messages would become disabled, no messages would be removed from E. By definition of the trivially-enabled-set, it follows that $m \in T$. □

Property 2. *Once a message is removed from E, it cannot rejoin E.*

Proof. If a message m_i is removed from E, that means a message m_{i+x}, where $x \geq \delta$, has appeared in $r_m{}'$. Since the presence of $m_{i+\delta}$ in $r_m{}'$ prevents m_i from being enabled, and messages are not removed from $r_m{}'$, m_i will never be enabled again. In other words, m_i will not be added to E again. □

Property 3. *A message cannot be removed from E unless it has appeared in $r_m{}'$.*

Proof. There is no message loss, so each low-level message must appear in r_m at least once. By Property 2, once a message is removed from E, it cannot rejoin E. Therefore, a message cannot be removed from E until it has appeared in $r_m{}'$, or else it will never appear and message loss will occur. □

Property 4. *A message cannot be removed from E unless it's in T.*

Proof. By Property 3, a message cannot be removed from E unless it has appeared in $r_m{}'$. By Property 1, if a message has appeared in $r_m{}'$ and is in E, it is also in T. Therefore a message cannot be removed from E unless it's in T. □

Property 5. *When $r_m{}'$ is empty, $|E| = \delta$.*

Proof. By the definition of enabled, all messages $m_0, m_1, ..., m_{\delta-1}$ are enabled when $r_m{}'$ is empty since the messages that would need to be present in $r_m{}'$ for them to be enabled do not exist. $m_0, m_1, ..., m_{\delta-1}$ total δ messages, so $|E| = \delta$ when $r_m{}'$ is empty. □

Property 6. *When $r_m{}'$ is empty, $|T| = \delta$.*

Proof. The message with the smallest subscript that would disable m_0 if it appeared in $r_m{}'$ is m_δ. The messages required to disable all other messages in E have subscripts greater than or equal to δ. Therefore, none of the messages in E when $r_m{}'$ is empty would cause any other messages in E to be disabled. So by definition of the trivially-enabled-set, all the messages in E are also in T when $r_m{}'$ is empty, so $|T| = |E| = \delta$ when $r_m{}'$ is empty. □

Property 7. *At any selected* $r_m{}'$, $|T| = \delta$.

Proof. For some selected $r_m{}'$:

$E = T \cup \{m_{i+y}, m_{i+y+1}, ..., m_{i+y+x-1}\}$

$|E| = y + x$

$T = \{m_i, m_{i+1}, ..., m_{i+y-1}\}$

$|T| = y$

Suppose in contradiction that $y < \delta$. Then $receive(m_{i+y})$ would not remove any messages from E, because it would disable messages with subscript $i + y - \delta < i$, none of which are in E. Therefore m_{i+y} would by definition be in T, creating a contradiction. Therefore y cannot be less than δ.

Suppose in contradiction that $y > \delta$. Then $receive(m_{i+y-1})$ would remove m_i from E. By definition of T, m_{i+y-1} would not be in T, creating a contradiction. Therefore y cannot be greater than δ.

Since y cannot be less than or greater than δ, $y = \delta$. □

Property 8. *At any selected* $r_m{}'$, $|E - T| \leq \delta$.

Proof. For some selected $r_m{}'$, we have the same conditions as in Property 7. Suppose in contradiction that $x > \delta$. Then $receive(m_{i+y+x-1})$ would at the least disable m_{i+y}. But $m_{i+y} \notin T$, and by Property 4, a message cannot be removed from E unless it's in T. Therefore we have a contradiction, and $x \leq \delta$. □

Property 9. $\delta \leq |E| \leq 2\delta$

Proof. Based on the properties of sets, $|E| = |T| + |E - T|$. From Property 7, $|T| = \delta$. From Property 8, $0 \leq |E - T| \leq \delta$. Therefore:

$0 \leq |E - T| \leq \delta$

$\delta \leq \delta + |E - T| \leq 2\delta$

$\delta \leq |T| + |E - T| \leq 2\delta$

$\delta \leq |E| \leq 2\delta$

So the minimum size of E is δ, and the maximum size is 2δ. □

3. IMPOSSIBILITY PROOFS RELATED TO THE RMTP WITH AN UNKNOWN REORDERING BOUND

If the reordering bound δ is known, then an algorithm can be developed that is dependent on knowing δ [5]. But what if δ is unknown? In the following section, we will prove that using the KJ Sender paired with any Receiver algorithm, an unknown δ makes the RMTP unsolvable. We refer to a particular guess for the value of δ as γ.

3.1 Impossibility Proof using the KJ Protocol Sender

Theorem 1. *If the channel exhibits unbounded but finite duplication, reordering bounded by an unknown bound δ, no loss, and no corruption, there exists no correct algorithm for the RMTP using the KJ Sender algorithm.*

Proof. Since the theorem specifies that the sender algorithm is the KJ Sender algorithm, we need to prove that a correct Receiver algorithm is impossible under these circumstances. We will prove this by contradiction. Suppose there is an algorithm in the Receiver that can produce the correct high-level output. Since we are using the KJ Sender, γ will act in place of δ in the KJ Sender code, and the bounded counters used for the low-level messages will be in the range $[0, 2\gamma]$. We examine two different executions of the algorithm that are possible when γ is equal to an arbitrary positive integer i, $\delta = 2i + 2$ and the high-level input is $M_0 M_1$ ... M_{2i+1}.

Execution 1: $r_m = m_0^0 m_1^1$ ... $m_{2i}^{2i} m_{2i+1}^0$

This low-level sequence is clearly possible under the given circumstances because none of the messages were received out-of-order. Since our hypothetical receiver algorithm is correct, the high-level output will be $M_0 M_1$ ... M_{2i+1}.

Execution 2: $r_m = m_{2i+1}^0 m_1^1$ ... $m_{2i}^{2i} m_0^0$

This low-level sequence is possible under the given circumstances because by the definition of δ, all of the low-level messages are enabled immediately. Since our hypothetical algorithm is correct, the high-level output will be $M_0 M_1$ ... M_{2i+1}.

However, there is a contradiction present between these two executions. The sequence of bounded counters received by the hypothetical Receiver algorithm is identical in both cases (it is $0, 1,$ ... $2i, 0$), but the actual sequence of messages is different. The transformation performed on the low-level sequence in Execution 1 to arrive at the correct high-level output is different from the transformation in Execution 2. In Execution 1, the messages are output in the order they are received, whereas in Execution 2, the first message that is received is output last, and vice versa. The only information that any Receiver algorithm can receive from the KJ Sender algorithm is a sequence of bounded counters and γ. If the sequence of counters is the same, and γ is the same, then the algorithm will behave in the same way. Therefore we have a contradiction, and Theorem 1 is proven. □

3.2 Introduction to the Tag Class of Algorithms

The KJ Sender is rather simple; it simply takes each high-level message, assigns the next bounded counter to the low-level message equivalent, and sends the low-level message through the channel. Next, we will develop a theorem similar to Theorem 1 for an expanded class of Sender algorithms. We will concern ourselves with Sender algorithms for which every low-level message consists of the information in a particular high-level message appended with some tag with bounded size. Any Receiver algorithm is allowed. However, like the KJ Sender, we do not allow for these Senders to make use of messages sent from the Receiver to the Sender. We will refer to this class of algorithms for the RMTP as the Tag Class. It should be noted that the KJ Sender is in the set of Tag Class Senders.

Sender algorithms in the Tag Class might not use bounded counters. However, we can

generalize the use of bounded counters as a number of bits attached to every low-level message, which we call a tag. Realistically, any tag attached to a low-level message will be represented as a sequence of B bits, and as such, can only take 2^B forms. Therefore the number of bits required to represent x different tags is $\lceil log_2 x \rceil$. We will refer to the number of bits in a tag as B, and the number of tags used as t.

Because of this expanded consideration, Theorem 1 does not apply to Senders in the Tag Class, because Theorem 1 relies on the Sender algorithm using bounded counters in a sequential manner as the KJ Sender does. For example, we don't know the tags that m_0 and m_{2i+1} will have appended to them, let alone that they share the same tag. We will have to develop another theorem to prove that no Tag Class algorithm can solve the RMTP with an unknown δ.

We don't know exactly how γ will be used in a particular Tag Class Sender, but we can imagine that the Tag Class Sender will either use γ to decide how many bits each tag will contain, or have this length hard-coded. Since the length of the high-level messages might not be constant, we must use the same number of bits for each tag, so that an algorithm always knows where the tag is located in a low-level message. Since the high-level messages do not have to be sent across the channel in order as the KJ Sender does, we refer to the sequence of low-level messages sent across the channel as $s_m = m_{l_0} m_{l_1}$ ... $m_{l_{n-1}}$ ... where m_{l_0} denotes the first low-level message sent by the Sender, m_{l_1} designates the second low-level message sent by the Sender, and so on. We should note that while a Tag Class Sender will send at least n low-level messages, one for each high-level message, it is possible that it will send more. However, we can ignore messages $m_{l_n} m_{l_{n+1}}$ ... for the purposes of the following lemmas and theorem without loss of generality.

3.3 Impossibility Proof for the Tag Class

Lemma 1. *For the Tag Class, the number of high-level messages input to the Sender cannot be communicated to the Receiver.*

Proof. The number of high-level messages input to the Sender cannot be communicated to the Receiver using a tag, because then the size of the tags would be unbounded, which violates the definition of the Tag Class. All low-level messages contain only a high-level message and a tag, therefore the number of high-level messages input to the Sender cannot be reliably communicated to the Receiver at all.

We can demonstrate this using a proof by contradiction. Suppose that there is some Tag Class Sender that can communicate the number of high-level messages to the Receiver. Since the size of the tags is bounded, the tag can only take 2^B forms. What if $2^B + 1$ high-level messages are sent? By the pigeonhole principle, the tag that tells the Receiver that there are $2^B + 1$ high-level messages must be identical to some tag that tells the Receiver that there are X high-level messages, where $1 \leq X \leq 2^B$. This contradicts the premise of our hypothetical algorithm, and thus Lemma 1 is proven. □

In the KJ Sender, since the number of bounded counters is calculated to be $2\gamma + 1$, the number of bounded counters is always odd. In terms of the Tag Class, t can be odd or even. We develop two lemmas for the Tag Class depending on whether t is odd or even.

Lemma 2. *For a given channel with unbounded but finite duplication, no loss, no corruption, and an unknown reordering bound δ, if t is odd, and if $t < 2\delta + 1$, then no Tag Class algorithm will work on this particular channel.*

Proof. We will prove by contradiction. Suppose we are using t tags, where t is odd, and the reordering bound of the channel is δ. Suppose that $t < 2\delta + 1$, which we can rearrange

to be $\delta > \frac{t-1}{2}$, and that there is an RMTP algorithm that will work without two-way communication. Let us examine two cases:

Case 1: input = $M_0 M_1$ ... M_t

$s_m = m_{l_0} m_{l_1}$ ... $m_{l_{\frac{t-1}{2}}} m_{l_{\frac{t-1}{2}}}$ ... $m_{l_{t-1}} m_{l_t}$ ...

Since there are $t+1$ high-level messages sent, there will be at least one pair of messages that share a tag in the first $t+1$ low-level messages sent. Without loss of generality, let's assume that m_{l_0} and m_{l_t} share a tag.

$r_m = m_{l_1}$ ... $m_{l_{\frac{t-1}{2}}} m_{l_0} m_{l_t} m_{l_{\frac{t+1}{2}}}$ ... $m_{l_{t-1}}$ ...

Since $\delta > \frac{t-1}{2}$, by the definition of δ it is possible for m_{l_0} to appear after $m_{l_{\frac{t-1}{2}}}$ and for m_{l_t} to appear before $m_{l\frac{t+1}{2}}$.

Case 2: input = $M_0 M_1$ ... M_{t-1}

$s_m = m_{l_0} m_{l_1}$ ... $m_{l_{\frac{t-1}{2}}} m_{l_{\frac{t+1}{2}}}$ ... $m_{l_{t-2}} m_{l_{t-1}}$ ...

Notice that one fewer high-level message was input.

$r_m = m_{l_1}$ ... $m_{l_{\frac{t-1}{2}}} m_{l_0} m_{l_0} m_{l_{\frac{t+1}{2}}}$ ... $m_{l_{t-1}}$ ...

Notice that the only difference in r_m between the two cases is that in Case 2, m_{l_0} is received again instead of m_{l_t}. But since m_{l_0} and m_{l_t} share a tag, the two cases have identical sequences of tags received by the Receiver. By Lemma 1, the Receiver does not know whether t or $t+1$ high-level messages were sent, and therefore cannot differentiate between the high-level input in Case 1 and the high-level input in Case 2. In Case 2, one of the m_{l_0} must be discarded in the Receiver algorithm, but in Case 1, neither m_{l_0} nor m_{l_t} should be discarded. Since the two cases result in identical tag sequences but different actions, there is a contradiction, and Lemma 2 is proven. □

When expressing tags in terms of bits, the maximum number of unique tags is 2^B. If the algorithm is going to make use of every available bit, then the number of tags will always be even, because all powers of two with positive integer exponents are even. In

this case, Lemma 2 does not apply, because it relied on the number of tags being odd. However, we can prove something very similar when the number of tags is even.

Lemma 3. *For a given channel with unbounded but finite duplication, no loss, no corruption, and an unknown reordering bound δ, if t is even, and if $t < 2\delta$, then no Tag Class algorithm will work on this particular channel.*

Proof. We will prove by contradiction. Suppose we are using t tags, where t is even, and the reordering bound of the channel is δ. Suppose that $t < 2\delta$, which we can rearrange to be $\delta > \frac{t}{2}$, and that there is an RMTP algorithm that will work without two-way communication. Let us examine two cases:

Case 1: input = $M_0 M_1$ ... M_t

$s_m = m_{l_0} m_{l_1}$ ... $m_{l_{\frac{t-2}{2}}} m_{l_{\frac{t}{2}}} m_{l_{\frac{t+2}{2}}}$ ... $m_{l_{t-1}} m_{l_t}$ ...

Since there are $t+1$ high-level messages sent, there will be at least one pair of messages that share a tag in the first $t + 1$ low-level messages sent. Without loss of generality, let's assume that m_{l_0} and m_{l_t} share a tag.

$r_m = m_{l_1}$ ... $m_{l_{\frac{t-2}{2}}} m_{l_{\frac{t}{2}}} m_{l_0} m_{l_t} m_{l_{\frac{t+2}{2}}}$ ... $m_{l_{t-1}}$ ...

Since $\delta > \frac{t}{2}$, by the definition of δ it is possible for m_{l_0} to appear after $m_{l_{\frac{t}{2}}}$ and for m_{l_t} to appear before $m_{l\frac{t+2}{2}}$.

Case 2: input = $M_0 M_1$ ... M_{t-1}

$s_m = m_{l_0} m_{l_1}$ ... $m_{l_{\frac{t-2}{2}}} m_{l_{\frac{t}{2}}} m_{l_{\frac{t+2}{2}}}$ ... $m_{l_{t-2}} m_{l_{t-1}}$ ...

Notice that one fewer high-level message was input.

$r_m = m_{l_1}$ ... $m_{l_{\frac{t-2}{2}}} m_{l_{\frac{t}{2}}} m_{l_0} m_{l_0} m_{l_{\frac{t+2}{2}}}$ ... $m_{l_{t-1}}$ ...

Notice that the only difference in r_m between the two cases is that in Case 2, m_{l_0} is received again instead of m_{l_t}. But since m_{l_0} and m_{l_t} share a tag, the two cases have identical sequences of tags received by the Receiver. By Lemma 1, the Receiver does not know whether t or $t + 1$ high-level messages were sent, and therefore cannot differentiate

between the high-level input in Case 1 and the high-level input in Case 2. In Case 2, one of the m_{l_0} must be discarded in the Receiver algorithm, but in Case 1, neither m_{l_0} nor m_{l_t} should be discarded. Since the two cases result in identical tag sequences but different actions, there is a contradiction, and Lemma 3 is proven. □

Theorem 2. *If the channel exhibits unbounded but finite duplication, reordering bounded by an unknown bound δ, no loss, and no corruption, there exists no correct Tag Class algorithm for the RMTP.*

Proof. Lemma 2 and Lemma 3 show that no matter how many unique tags are used in a Tag Class algorithm (i.e. no matter the value of t), if t is sufficiently small in relation to the reordering bound, then no algorithm will work. Although an algorithm could work on a particular channel if there are enough tags, there could exist a different channel with a large enough reordering bound to render the number of tags too small. □

3.4 Introduction to Backchannel Class and the Adjustment Interface

We will now examine another class of algorithms that is a superset of the Tag Class. This class, which we will call the Backchannel Class, allows for two-way communication; that is, the Receiver is allowed to send messages to the Sender, which can then act upon the information in those messages. We also allow for different types of messages to be sent between the Sender and Receiver. In addition the format of low-level messages in the Tag Class, low-level messages in the Backchannel Class can contain other information that does not pertain to a high-level message. However, the algorithms in the Backchannel Class are not permitted to send partial high-level messages; high-level messages can only be sent as the whole message appended with some tag.

The most obvious use for two-way communication with regards to working with an unknown δ is for the Receiver to try to detect some minimum value of δ, and based on that, determine whether the Sender used enough unique tags to be able to solve the RMTP.

If too few tags were used, the Receiver could send a message to the Sender that the number of tags needs to be increased. The Sender could then start appending its messages with larger tags to accommodate the greater number of unique tags, and inform the Receiver of the new size.

Note that this hypothetical method would not necessarily guarantee a totally correct algorithm, but over time, the algorithm would "tend towards correctness" as the number of tags is adjusted again and again. The only way that the output could be guaranteed to be correct is if at some point the Sender knows that the high-level input is complete and thus has knowledge of the entire sequence of high-level messages (in other words, the Sender knows that a particular $SEND$ event is the final $SEND$ event). It is also important to realize that any messages that the Receiver sends to the message, as well as any that the Sender sends to the Receiver concerning a new tag size, are subject to the same undesirable channel behavior that every other message is.

We define "tending towards correctness" as follows. If a Backchannel algorithm tends towards correctness, then for an infinite input sequence, the input and output sequences share some infinite suffix. In other words, at some point in the input sequence, all subsequent messages will appear in the correct order beginning at some point in the output sequence.

An algorithm that uses two-way communication in this way can be encapsulated with an interface on top of the Sender and Receiver. We will call this interface the Adjustment Interface. Both the Sender and Receiver contain a variable for t, the number of unique tags currently in use, as well as for γ, the assumed value of δ. The Receiver contains a variable called tagQuality, which is initially set to "unknown." As the Receiver gets low-level messages, at some interval will run the Analyze() routine, which examines the tags of the messages that have been received (in other words, ${r_m}'$) to detect a tag that contradicts γ. If a contradiction to γ is detected, tagQuality is set to "bad", the Receiver

increases t and γ by some amounts, and the Receiver sends a "BAD" message to the Sender. Then tagQuality is reset to "unknown." Upon receipt of the "BAD" message, the Sender increases t and γ by the same amounts and resumes operation with a new number of unique tags. Any implementation of the Adjustment Interface must account for the channel's undesirable behavior.

We will show that if the number of tags is lower than the reordering bound itself, no Backchannel algorithm can rely on the Adjustment Interface to tend towards correctness. Note that this is a tighter restriction of the number of tags compared to δ than those considered in the Tag Class lemmas.

3.5 Impossibility Proof for the Backchannel Class with Adjustment Interface

Theorem 3. *If the channel exhibits unbounded but finite duplication, reordering bounded by an unknown bound δ, no loss, and no corruption, no Backchannel algorithm can rely on the Adjustment Interface to tend towards correctness.*

Proof. Suppose there is a Backchannel Receiver that can correctly determine when it is necessary to set tagQuality to "bad." In other words, the Receiver's Analyze() routine works. We examine the scenario in which $\delta > t$ and the high-level input is $M_0M_1...M_t$. The proof depends on an integer relative to t that is expressed differently based on whether t is odd or even. We will call this value t_{half}. When t is even, $t_{half} = \frac{t}{2}$. When t is odd, $t_{half} = \frac{t-1}{2}$.

If a message appears t_{half} or more places out of order (meaning that $\delta > t_{half}$), then Analyze() sets tagQuality to "bad". This is because by Lemmas 2 and 3, if $\delta > t_{half}$, there are not enough tags to correctly solve the RMTP. The low-level sequence, r_m, is at least $t + 1$ messages long since $t + 1$ high-level messages were sent. By the definition of δ, the first $t + 1$ messages are trivially enabled. Since there are t possible tags, but at least $t + 1$ messages, there will be at least one pair of low-level messages that share a tag in the first

$t+1$ messages. Note that we are referring to a sequence of low-level messages before they are sent through the channel (s_m), not after (r_m).

No matter which pair of messages share a tag, there are at least two possible r_m that have the same sequence of tags but from which a correct Analyze() routine would draw different conclusions. In other words, a correct Adjustment Interface would set tagQuality to "bad" for one possible r_m but would not for the other possible r_m. The function below, called $ProduceContradiction$, can produce two such r_m for any pair of messages in s_m that share a tag. We will refer to the pair of messages that share a tag as the sharing pair.

In the function, $SeqOK$ denotes the r_m for which the Adjustment Interface's Analyze() routine does not change the value of tagQuality. $SeqBad$ denotes the r_m for which the Analyze() routine sets tagQuality to "bad". Note that the parameters x and y are positions of the message in s_m, and are therefore determined by the Sender algorithm, not the channel's behavior. A precondition of the function is that $0 \leq x < y \leq t$.

ProduceContradiction(x, y):

1: $Seq = m_{l_0} m_{l_1}$ ... $m_{l_{t-1}} m_{l_t}$ //this is s_m

2: **if** $(y - x < t_{half})$ **then** //we must reposition one of the messages in the sharing pair so that they are at least t_{half} spaces apart

3: **if** $(y + t_{half} - 1 > t)$ **then** //if m_{l_y} is too close to the end of s_m, reposition m_{l_x}

4: $SeqOK =$ swap $m_{l_{x-t_{half}+1}}$ and m_{l_x} in Seq

5: **else** //reposition m_{l_y} to be at least t_{half} spaces away from m_{l_x}

6: $SeqOK =$ swap m_{l_y} and $m_{l_{y+t_{half}-1}}$ in Seq

7: **end if**

8: **else** //if the sharing pair are at least t_{half} spaces apart, swapping their positions will show that $\delta > t_{half}$

9: $SeqOK = Seq$

10: **end if**

11: $SeqBad$ = swap m_{l_x} and m_{l_y} in $SeqOK$ //this swap causes a message in $SeqBad$ to be at least t_{half} places out of order

12: **return** $SeqOK$ and $SeqBad$

end function

The intuition for the function $ProduceContradiction$ can be explained as follows. If the sharing pair are at least t_{half} spaces apart in s_m, then if the channel swaps their positions for r_m ($SeqBad$), the Analyze() will know that messages appeared t_{half} or more places out of order. Therefore $\delta > t_{half}$ and tagQuality should be set to bad. However, if the channel does not exhibit any bad behavior for this input and r_m equals s_m ($SeqOK$), tagQuality should not be set to bad because Analyze() cannot tell that $\delta > t_{half}$. The contradiction arises based on the fact that the r_m in these two cases have the same sequence of tags since the only difference between them is the sharing pair swapping positions.

If the sharing pair are less than t_{half} spaces apart, it is a little trickier to show that a contradiction is possible. The key is that it is always possible for the sharing pair to be repositioned by the channel to be at least t_{half} spaces apart by only moving one of the messages fewer than t_{half} spaces. The r_m for which this repositioning is the only change from s_m ($SeqOK$) does not indicate to Analyze() to set tagQuality to "bad" because the reordering that takes place is not t_{half} or greater. If in another r_m ($SeqBad$) the sharing pair are in swapped positions from the repositioned r_m ($SeqOK$), Analyze() must set tagQuality to "bad" since the message that was swapped in the repositioning but is not part of the sharing pair has a position in s_m that is at least t_{half} different from the position in s_m of the message in the sharing pair that was not repositioned. Once again, the contradiction arises based on the fact that the r_m in these two cases have the same sequence of tags since the only difference between them is two messages with the same tag swapping positions.

Since a potential contradiction is unavoidable regardless of whether t is odd or even, we conclude that our hypothetical Receiver algorithm that has a working Analyze() routine that can reliably set tagQuality to "bad" does not exist.

Since $t < \delta$, it follows that $t < 2\delta$ and $t < 2\delta + 1$. Based on Theorem 2, if $t < 2\delta + 1$ for an odd t, or $t < 2\delta$ for an even t, then a Tag Class algorithm cannot work. If tagQuality is never set to "bad", then no two-way communication will take place, so the algorithm will functionally behave as a Tag Class algorithm. By these inequalities, and the fact that the algorithm will behave as a Tag Class algorithm, the algorithm will continue to produce incorrect output even for an infinite input sequence and will not tend towards correctness.

Therefore if $t < \delta$, then the Adjustment Interface cannot be relied upon, and the Backchannel algorithm may remain incorrect without ever tending towards correctness. Since there is no way to guarantee that the channel will not have $t < \delta$, Theorem 3 is proven. □

4. AN IMPLEMENTATION OF THE ADJUSTMENT INTERFACE GIVEN A RANGE FOR THE REORDERING BOUND

4.1 Background

Although there is currently no known method for solving the RMTP with a totally unknown reordering bound, it is possible to implement the Adjustment Interface and tend towards correctness if the unknown reordering bound lies within a known range; i.e., we know that $a \leq \delta \leq b$, where a and b are known positive integers.

In order to implement the Adjustment Interface, we need a method for setting tagQuality to "bad" at the appropriate time. This algorithm accomplishes this using a data structure in the Receiver called $expected\_enabled$. This data structure is a map that contains (tag, bool) key-value pairs, and represents the enabled-set of tags if the guess at the reordering bound is correct. Using a guess at the reordering bound (γ), and the number of unique tags in use (t), the $ANALYZE()$ routine will update $expected\_enabled$ after every tag that is received from the channel. If the received tag is not in $expected\_enabled$, then tagQuality is set to "bad", since clearly the reordering bound guess is incorrect if a tag is received that would not have been enabled if the guess were correct.

Given $a \leq \delta \leq b$, if b is not too much larger than a, the RMTP could be solved by assuming the worst case scenario, that $\delta = b$. This is because by Theorem 2, if t is great enough to solve the RMTP when $\delta = b$, then $t \geq 2b$, and $2b$ is greater than the number of tags needed for any value of δ that is less than b. Therefore, this algorithm is situationally useful, where the user wants to minimize the size of tags appended to the low-level messages. Given a known range that δ can fall within, the algorithm will guess the lowest possible value of δ so that if this guess is correct, the tag size in bits is the

smallest it can be.

But how should the algorithm select the number of tags to use initially? As we showed in Theorem 3, the number of tags t must be at least as large as the reordering bound δ. However, this does not necessarily mean that if $\delta \leq t < 2\delta$, then the Adjustment Interface will always work.

This implementation of the Adjustment Interface will not work if it is at all possible for a tag to be in $expected\_enabled$, and at the same time, for two different messages with that tag to be enabled in reality. The $expected\_enabled$ data structure will change in different ways depending on which of those two messages will be received, and it is impossible to reliably predict which one will be received. It is valid for there to be two messages with the same tag in E if that tag is not in $expected\_enabled$, because then no matter which of the two messages is received, tagQuality will be set to "bad", and $expected\_enabled$ becomes irrelevant until the number of tags has been increased.

For every available space in the real enabled-set (E) in excess of the number of tags, it is possible that a duplicate tag occupies that space. Therefore it is possible that E can contain up to $MaxSize(E) - t$ duplicate tags at once. The difference between the number of tags and the maximum size of $expected\_enabled$ needs to be at least $MaxSize(E) - t$, because there need to always be at least that many tags that cannot be in $expected\_enabled$. An implementation of the Adjustment Interface in this manner can then ensure that the tags that are duplicate in E are always the tags that are absent from $expected\_enabled$. We can express this with the inequality:

$t - MaxSize(expected\_enabled) \geq MaxSize(E) - t$

We know from Property 9 that the maximum size of E is 2δ, and therefore the maximum size of $expected\_enabled$ is 2γ. So the inequality can be rearranged to:

$t - 2\gamma \geq 2\delta - t$

$2t \geq 2\delta + 2\gamma$

$t \geq \delta + \gamma$

Initially our guess γ is the lowest possible value of δ, but we need to account for the possibility that δ is in fact its highest possible value. Therefore, given $a \leq \delta \leq b$, we set our initial number of tags to be $t = \delta + \gamma = b + a$.

For example, given $10 \leq \delta \leq 20$, the initial number of tags would be $t = 20+10 = 30$. It only takes 5 bits to express 30 unique tags. If δ falls in the range $[13, 20]$, then the Adjustment Interface implementation will eventually increase the number of tags to the range $[33, 40]$, which will require 6 bits. However, if used on channels for which δ falls in the range $[10, 12]$, the number of tags used once $\gamma = \delta$ will be in the range $[30, 32]$, requiring only 5 bits and saving a bit per low-level message that would be used if δ had been assumed to be the highest possible value, 20, from the onset. It is not possible to save more than one bit of space per message using this algorithm, because the ratio of the largest and smallest possible values of t, which is $\frac{2b}{a+b}$, cannot be greater than 2, and increasing t by a factor of 2 requires using one more bit. Therefore, the algorithm may be most useful when the message contents are very small and a very large number of messages must be stored.

4.2 High-Level Explanation of the Algorithm

This algorithm that implements the Adjustment Interface is heavily based on the KJ Protocol [5]. We provide pseudocode below for the Sender, Channel, and Receiver algorithms, as well as some procedures that the Receiver algorithm executes. The constants $delta\_min$ and $delta\_max$ are the bounds on the known range of δ, and are known by the Sender and Receiver. Instances of δ in the Sender and Receiver have been replaced with the guess, γ. The variable Aux_A is an auxiliary counter attached to each message that is useful for analyzing and testing the algorithm. However, no part of the algorithm depends on the value of Aux_A, and a true implementation of the algorithm that has a bound on the

message size would not use them. One key difference between this algorithm and the KJ Protocol is that in addition to a bounded counter, a message's tag also consists of t, the number of unique tags in use when the message was sent across the channel. This t can be considered a generation identifier for the low-level messages, if all the messages sent between each correction of γ are considered a generation.

The $SEND$ event is mostly unchanged from the KJ Protocol. A high-level message is converted to a low-level message by appending to the message a bounded counter and the number of tags currently in use. The low-level message is entered into the $send\_pending$ queue, and the bounded counter and auxiliary counter variables are incremented. The $send$ event as an output event for the Sender is unchanged from the KJ Protocol; the head of $send\_pending$ is removed and sent across the channel. The $receive$ event as an input event to the Sender is new. The message received by the Sender only consists of an integer that represents the new guess at the reordering bound. If this number is greater than the Sender's current guess γ, then γ and the number of tags are increased, the bounded counter variable is reset, and all the messages in $send\_pending$ are updated with the new bounded counters (which will have one more unique value than before). The reordering and duplication of these messages sent from the Receiver to the Sender is not problematic, since the value of γ will only ever increase, so larger values that are received by the Sender are known to be more recent.

The Channel algorithm is unchanged from the KJ Protocol. However, this algorithm uses two Channel algorithms. One Channel has input events that are the Sender's output events and output events that are the Receiver's input events. The other Channel has input events that are the Receiver's output events and output events that are the Sender's input events.

The Receiver has undergone the most significant changes from the KJ Protocol, since most of the Adjustment Interface implementation is in the Receiver. In the $receive$ in-

put event to the Receiver, if the received low-level message has an older t value, then it is placed at the end of the $lower\_tagged$ queue. Otherwise, it is analyzed by the $ANALYZE$ routine to see if the bounded counter of the message contradicts the assumed value of the reordering bound, γ. If γ is not proven false, then the $expected\_enabled$ data structure is updated by removing tags that have been disabled by the receipt of the newest low-level message and by inserting tags that have been enabled by the receipt of the newest low-level message. Then the algorithm attempts to insert the message into the $receive\_pending$ array; the code that attempts this insertion is taken from the KJ Protocol. If γ is proven false, i.e. the bounded counter is not in the $expected\_enabled$ data structure, then the message is placed at the end of the $lower\_tagged$ queue, but not before resetting most of the Receiver's local variables and data structures. Also at this point, γ and the number of unique tags are increased, and a flag is set that enables the Receiver's $send$ output event. This $send$ event simply sends the current back to the Sender, and the flag is disabled.

Finally, the $RECEIVE$ event is similar to in the KJ Protocol. The difference is that if the $lower\_tagged$ queue contains any messages, these have priority and are converted to high-level messages first. This means that once a value of t is found to be too low, no attempt is made to fix the channel's duplication and reordering of low-level messages with that value of t. The high-level output will probably be incorrect until messages with the newest value of t begin to be received. This process will continue until $\gamma = \delta$, at which all subsequent high-level output will be correct. In other words, the algorithm tends towards correctness, as described in the earlier section. If the $lower\_tagged$ queue is empty, then the $RECEIVE$ event behaves as in the KJ Protocol, only outputting the head of $receive\_pending$ if its bounded counter matches the Receiver's bounded counter variable, and incrementing that variable if the message is output.

4.3 Pseudocode for the Algorithtm

Algorithm 1: Algorithm for the Sender

1: $counter_A$, an integer, initially 0

2: $send\_pending$, a FIFO queue, initially empty

3: Aux_A, an integer, initially 0

4: γ, an integer, initially $delta\_min$

5: $numTags$, an integer, initially $delta\_min + delta\_max$

6: **input events:**

7: **event** $SEND(m)$

8: **effects:**

9: $send\_pending$.enq$((m, counter_A, numTags, Aux_A)$

10: $counter_A \leftarrow (counter_A + 1)$ mod $numTags$

11: $Aux_A \leftarrow Aux_A + 1$

12: **end event**

13: **event** $receive(c)$

14: **effects:**

15: **if** $c > \gamma$ **then**

16: $\gamma \leftarrow c$

17: $numTags \leftarrow \gamma + delta\_max$

18: $counter_A \leftarrow 0$

19: **for** $i \leftarrow 0$ to $|send_pending| - 1$ **do**

20: $send\_pending$.deq$((m, *, *))$

21: $send\_pending$.enq$((m, counter_A, numTags))$

22: $counter_A \leftarrow (counter_A + 1)$ mod $numTags$

23: **end for**

24: **end if**

25: **end event**

26: **output events:**

27: **event** $send(m, c, t, a)$

28: **preconditions:**

29: (m, c, t, a) is at the head of $send\_pending$

30: **effects:**

31: remove (m, c, t, a) from $send\_pending$

32: **end event**

Algorithm 2: Algorithm for the Channel

1: $in\_transit$, array, initially empty

2: **input events:**

3: **event** $send(m)$

4: **effects:**

5: insert $(m, False)$ at the end of $in\_transit$

6: **end event**

7: **output events:**

8: **preconditions:**

9: $(m, y) \in in\_transit$ at some index l for some y

10: for all $(m', y') \in in\_transit$ with index $\leq l - \delta$, $y' = True$

11: **effects:**

12: remove from $in\_transit$ all entries with index $\leq l - \delta$

13: $y \leftarrow True$

14: **end event**

Algorithm 3: Algorithm for the Receiver

1: $counter_B$, an integer, initially 0

2: $receive\_pending$, array, initially empty

3: $delivered$, a FIFO queue, initially empty

4: $lower\_tagged$, a FIFO queue, initially empty

5: γ, an integer, initially $delta\_min$

6: $numTags$, an integer, initially $delta\_min + delta\_max$

7: $sendBackNeeded$, a boolean, initially $False$

8: $expected\_enabled$, a map, initially contains all key-value pairs ($i \mapsto False$ for all $i \leftarrow 0$ to $delta\_min - 1$

9: **input events:**

10: **event** $receive(m, c, t, a)$

11: **effects:**

12: **if** $t < numTags$ **then**

13: $lower\_tagged$.enq((m, c, t, a))

14: **else if** $ANALYZE(c)$ **then**

15: $ATTEMPTADDTORP(m, c, t, a)$

16: **else**

17: $lower\_tagged$.enq((m, c, t, a))

18: **end if**

19: **end event**

20: **output events:**

21: **event** $send(c)$

22: **preconditions:**

23: $sendBackNeeded = True$

24: **effects:**

25: $sendBackNeeded \leftarrow False$

26: **end event**

27: **event** $RECEIVE(m)$

28: **preconditions:**

29: $((m, c, t, a)$ is at the head of $receive\_pending \wedge c = counter_B) \vee (m, c, t, a)$ is at the head of $lower\_tagged$

30: **effects:**

31: **if** $|lower\_tagged| > 0$ **then**

32: remove (m, c, t, a) from $lower\_tagged$

33: **else**

34: remove (m, c, t, a) from $receive\_pending$

35: $counter_B \leftarrow (counter_B + 1)$ mod $numTags$

36: $delivered$.enq$((m, c, t, a))$

37: **if** $|delivered| = \gamma + 1$ **then**

38: $delivered$.deq()

39: **end if**

40: **end if**

41: **end event**

Algorithm 4: Procedures used by the Receiver in the $receive$ event

procedure $ANALYZE(c)$
if $(c \mapsto False) \in expected\_enabled$ **then**
    $expected\_enabled[c] \leftarrow True$
    remove all keys i from $expected\_enabled$, where i is a tag that should no longer be enabled
    insert all key-value pairs $(j \mapsto False)$ in $expected\_enabled$, where j is a tag that should now be enabled
    return $True$
else if $(c \mapsto *) \notin expected\_enabled$ **then**
    $sendBackNeeded \leftarrow True$
    $counter_B \leftarrow 0$
    $\gamma \leftarrow \gamma + 1$
    $numTags \leftarrow \gamma + delta\_max$
    $expected\_enabled$.clear()
    insert all key-value pairs $(i \mapsto False)$ in $expected\_enabled$, for all $i \leftarrow 0$ to $\gamma - 1$
    $delivered$.clear()
    $lower\_tagged \leftarrow lower\_tagged$ concat $receive\_pending$
    $receive\_pending$.clear()
    return $False$
end if
end procedure
procedure $ATTEMPTADDTORP(m, c, t, a)$
$total\_msgs \leftarrow delivered$ concat $receive\_pending$
if $(*, c, *) \notin total\_msgs$ **then**

23: $\alpha \leftarrow \gamma$

24: **else**

25: $\alpha \leftarrow$ number of entries after most recent occurrence of $(*, c, *)$ in $total\_msgs$

26: **end if**

27: **if** $\alpha \geq \gamma$ **then**

28: $ADDTORECEIVEPENDING(m, c, t, a)$

29: **end if**

30: **end procedure**

31: **procedure** $ADDTORECEIVEPENDING(m, c, t, a)$

32: **if** $|receive\_pending| = 0$ **then**

33: insert (m, c, t, a) in $receive\_pending$

34: **return**

35: **end if**

36: $current \leftarrow$ tag of the most recent message in $receive\_pending$

37: **if** $MLT(current, c)$ **then**

38: insert (m, c, t, a) at the end of $receive\_pending$

39: **return**

40: **end if**

41: **for** $i \leftarrow |receive\_pending| - 1$ to 1 **do**

42: $current \leftarrow$ tag of the message at index i in $receive\_pending$

43: $next \leftarrow$ tag of the message at index $i - 1$ in $receive\_pending$

44: **if** $MLT(next, c) \wedge MLT(c, current)$ **then**

45: insert (m, c, t, a) in $receive\_pending$ between the messages at indices $i - 1$ and i

46: **return**

47: **end if**

48: **end for**

49: $next \leftarrow$ tag of the message at index 0 in $receive\_pending$

50: **if** $MLT(c, next)$ **then**

51: insert (m, c, t, a) in $receive\_pending$ at index 0

52: **return**

53: **end if**

54: **end procedure**

55: **procedure** $MLT(c_1, c_2)$

56: **if** $c_1 < c_2$ **then**

57: **return** $c_2 - c_1 \leq \gamma$

58: **end if**

59: **return** $c_1 - c_2 > \gamma$

60: **end procedure**

5. CONCLUSION AND FURTHER RESEARCH

Prior to this paper, analysis and study of the RMTP with a channel that exhibits bounded reordering and unbounded but finite duplication assumed that the bound on reordering was known. The theorems and proofs in this paper show that with various general classes of algorithms, the RMTP cannot be solved under these circumstances with an unknown reordering bound. This paper also provides an algorithm that extends the KJ Protocol [5], and given a known range of values between which the reordering bound can lie, can use the minimum amount of memory per message required to solve the RMTP, and can increase the size of the message tags if necessary.

Further research in this area could involve several tasks, including developing a correctness proof for the algorithm implementation of the Adjustment Interface; an improvement, expansion, or optimization of the aforementioned algorithm; or a general impossibility proof that proves that the RMTP cannot be solved with an unknown reordering bound under any circumstances. I strongly believe that this last task is possible, and may be simpler than I have long suspected. If the low-level message sequences can be mapped to certain high-level outputs, without regard to the contents of the low-level messages, and a contradiction can be shown (i.e. a low-level sequence produces a certain output but should produce another to be correct), then it may be proven that no algorithm can solve the RMTP with an unknown reordering bound.

REFERENCES

[1] N. V. Stenning, "A data transfer protocol," *Computer Networks*, vol. 1, no. 2, pp. 99–110, 1976.

[2] R. A. S. K. A. Bartlett and P. T. Wilkinson, "A note on reliable full-duplex transmission over half-duplex links," *Communications of the ACM*, vol. 12, pp. 260–261, May 1969.

[3] A. F. M. F. N. L. Y. M. D.-W. W. Y. Afek, H. Attiva and L. Zuck, "Reliable communication over unreliable channels," *Journal of the ACM debug hello*, vol. 41, pp. 1267–1297, November 1994.

[4] D.-W. Wang and L. D. Zuck, "Tight bounds for the sequence transmission problem," *Proceedings of the Eighth Annual ACM Symposium on Principles of Distributed Computing*, pp. 73–83, 1989.

[5] K. D. Ortiz-Lopez and J. Welch, "Bounded protocols for efficient reliable message transmission," *IEEE International Parallel and Distributed Processing Symposium*, pp. 1–36, August 2017.