Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Technical Report Nr. 719

Systems Group, Department of Computer Science, ETH Zurich

E-Cast: Elastic Multicast

by

Philipp Unterbrunner
Gustavo Alonso
Donald Kossmann

February 24, 2011

inf | Informatik
Computer Science

Abstract

This technical report describes E-Cast: a uniform causal-total-order multicast protocol designed to implement fault tolerant, highly elastic, yet strongly consistent database engines in the cloud. In contrast to traditional group communication, the model underlying E-Cast defines multicast as a stateful routing problem. In this document, we provide a rigorous formalization of the routing problem, show how partial replication with strong consistency guarantees can be reduced to said routing problem, and present its efficient algorithmic solution, E-Cast.

1 Introduction

Elasticity in a cloud infrastructure refers to the ability to dynamically change the amount of computing resources needed as demand dictates [5]. For software to be considered elastic, it must be able to expand or contract in terms of the number of nodes where it runs, in response to changes in the computing environment (e.g., load, failures, multi-tenancy, etc.). In the case of databases, elasticity implies the ability to dynamically spawn or withdraw additional storage nodes, where copies of the data are placed to provide additional storage or processing capacity. For performance reasons, these nodes should not contain complete copies of the data but only partial replicas, which reduces the storage as well as the query and update overhead [26, 31, 32].

Existing solutions for database replication tend to assume a static (hence inelastic) configuration. Middleware database replication solutions use different agreement protocols to ensure that all updates are propagated and applied in the same order at all storage nodes. In most cases, they use a state-machine approach or atomic broadcast [15, 35, 39] over all storage nodes in the system, thereby limiting the solution to full replication [19, 29, 36].

There have been attempts to improve on such systems through the use of partial replication and group communication [2, 18, 20, 25]. Group communication is a combination of communication service (reliable, ordered multicast primitives) and membership service [8, 12]. The membership service establishes consensus on a sequence of system views (configurations), each of which defines one or more process groups. Processes send messages within and (depending on the protocol) across views and groups, and the communication service gives ordering and delivery guarantees for these messages. It is generally assumed that membership changes are infrequent, and it is deemed acceptable that they cause a significant performance hiccup and/or consistency degradation when they take place [7, 24].

Implementing partial replication over large numbers of data objects and storage nodes through group communication implies a potentially large number of groups (copies of items will be placed in different subsets of nodes), and also frequent changes to the composition of those groups as the system scales up and down and nodes appear and disappear (due to node failures or simply because the nodes are assigned to other tasks).

To support such challenging scenarios, we propose a novel multicast primitive, *E-Cast*, specifically tailored to provide strong delivery guarantees across many groups and frequent membership changes. The key observation behind E-Cast is that it is possible to determine the destination processes of a message exclusively by looking at the past history of messages in the system rather than relying on a membership protocol. Based on an *application-specific function* defined over the sequence of messages delivered so far, the protocol determines where a message has to be delivered.

E-Cast combines the advantages of state-machine replication and group communication. As in reconfigurable state machines, the system state is assumed to include configuration state, removing the membership service needed in group communication. As a result, E-Cast is easy to reason about and use. But unlike reconfigurable state machines, E-Cast gives applications the freedom to choose which

message to deliver to which process, thus being efficient and scalable like group communication. E-Cast multicasts messages in a manner which involves only the sender process, the destination processes, and some (small) number of router processes. Furthermore, the number of messages that E-Cast can pipeline is only bounded by the computational resources of the individual processes. These two protocol properties translate to scalability and asynchronous reconfiguration, the features needed to build elastic database engines in the cloud.

In this document we formalize the stateful routing problem underlying E-Cast, and show how partial replication with strong consistency guarantees can be expressed as said routing problem. We present the E-Cast protocol and prove that it is a correct solution of the routing problem.

1.1 Document Structure

The rest of the document is organized as follows. Section 2 discusses related work. Section 3 gives a formal problem statement. Section 4 presents the E-Cast protocol in pseudo-code. Section 5 describes a generic reduction of partial replication to E-Cast. Section 6 compares E-Cast to the state-machine approach and group communication. Section 7 concludes the document. A proof of correctness and quiescence of E-Cast is provided in the appendix.

2 Related Work

Reliable, ordered multicast protocols and their efficient implementation have been an active research topic for several decades. Classic examples of multicast toolkits include Isis [9], Transis [17], Horus [38], and Totem [3]. More recent work includes QuickSilver [27], JGroups [16], and Spread [4]. Birman [6] gives a historic overview, and Defago et. al. [15] provide an excellent survey of various protocols and their formal properties. Like many of these group communication protocols and toolkits, E-Cast has been designed for partial replication. But instead of using process groups, E-Cast models partial replication as a stateful routing problem, and uses state-machine replication [30, 35] to implement an infallible layer of *router* processes. According to Defago et. al. [15], E-Cast can be classified as a "unicast-broadcast fixed sequencer" protocol, where the sequencer is a replicated, reconfigurable state-machine.

In [6] and [28], Birman, Ostrowski et. al. describe the vision of *Live Distributed Objects*, which implement massive multiplayer games or military simulations without a simulation server. Instead, data objects are replicated and simulated at the clients. E-Cast is well-suited for these scenarios. Furthermore, systems such as G-Store [14] have recently begun to emerge, which add strong consistency guarantees on top of specific cloud storage systems. In this document, we explain how E-Cast can be used to implement strongly consistent partial replication *independent* of a specific data model or storage system.

3 Problem Statement

To overcome the scalability limits of atomic broadcast, various models of *multicast* have been formalized in the past [6, 12, 15]. The model of E-Cast differs from previous work in one crucial aspect: there is no explicit notion of *process group*. Instead, groups are implicit in message routing, which is determined—exclusively—through an application-specific function of the history of delivered messages[1]. We now present a rigorous formalization of the resulting routing problem.

[1] Not quite true, since some messages may not be delivered, but true enough.

3.1 System Model and Definitions

The system consists of a set of processes P, which are sequences of discrete events. Processes submit and deliver messages. Submitting or delivering a message is an event. Let M be the universe of messages. Define $M^\Sigma \subseteq M$ as the set of submitted messages, and $M^\Delta \subseteq M$ as the set of delivered messages. Every process $p \in P$ submits a set of messages $M_p^\Sigma \subseteq M^\Sigma$, and delivers a set of messages $M_p^\Delta \subseteq M^\Delta$.

It is assumed that every message $m \in M^\Sigma$ is submitted exactly once[2]. Let $\text{sub}(m)$ denote the unique submission event for $m \in M^\Sigma$, and $\text{Dlv}(m,p)$ denote the set[3] of delivery events for $p \in P$ and $m \in M_p^\Delta$. The causal order relation $e \to e'$ over events e, e' is defined as the minimal, transitive relation that holds if both e and e' are events of some process p and e precedes e' in the execution of p, or $e = \text{sub}(m)$ and $e' \in \text{Dlv}(m,p)$ for some $p \in P$ and some $m \in M$.

We refer to a sequence of messages as an *m-seq*. Let $\mathbb{Q}$ be the universe of m-seqs over M. An m-seq $Q \in \mathbb{Q}$ is a pair $Q = \langle M_Q, \rightarrowtail_Q \rangle$, where $M_Q \subseteq M$ is a set of messages, and $\rightarrowtail_Q$ is a strict total order relation over M_Q. A function $\text{prefix}(Q,m)$ is defined, where Q is an m-seq, and m is a message. If $m \in M_Q$, the function yields the prefix of Q up to but excluding m. If $m \notin M_Q$, then $\text{prefix}(Q,m)$ is Q itself.

The existence of a single, application-specific function $\text{destset}(Q,m)$ is assumed, which maps every pair of m-seq Q and message m to a finite set of processes $\text{destset}(Q,m) \subseteq P$ called the *destination set* of m according to Q. We write $\text{destsetpfx}(Q,m)$ as a shorthand for $\text{destset}(\text{prefix}(Q,m),m)$. The existence of the destset function distinguishes the proposed model from traditional state-machine replication and group communication.

A process is considered *correct* iff it is not faulty. In this document, we only consider crash failures (no Byzantine failures), but the problem statement is independent of the failure model.

3.2 Predicates over Message Sequences

The following predicates over m-seqs are defined.

valid(Q). *An m-seq $Q = \langle M_Q, \rightarrowtail_Q \rangle$ is* valid *iff every message in M_Q was submitted, and no message is delivered that is not in M_Q. Formally,*

$$\text{valid}(\langle M_Q, \rightarrowtail_Q \rangle) \stackrel{\text{def}}{=} M^\Delta \subseteq M_Q \subseteq M^\Sigma$$

minimal(Q). *An m-seq $Q = \langle M_Q, \rightarrowtail_Q \rangle$ is* minimal *iff every process that delivers a message m is in the destination set of m according to Q. Formally,*

$$\text{minimal}(\langle M_Q, \rightarrowtail_Q \rangle) \stackrel{\text{def}}{=} \forall p \in P,\ m \in M_p^\Delta : p \in \text{destsetpfx}(Q,m)$$

complete(Q). *An m-seq $Q = \langle M_Q, \rightarrowtail_Q \rangle$ is* complete *iff for every message $m \in M_Q$, every correct process in the destination set of m according to Q delivers m. Formally,*

$$\text{complete}(\langle M_Q, \rightarrowtail_Q \rangle) \stackrel{\text{def}}{=} \forall m \in M_Q, p \in \text{destsetpfx}(Q,m) : \text{correct}(p) \Longrightarrow m \in M_p^\Delta$$

gap-free(Q). *An m-seq $Q = \langle M_Q, \rightarrowtail_Q \rangle$ is* gap-free *iff for every message $m \in M_Q$ submitted by some process p, every message m' submitted previously by p is also in M_Q. Formally,*

$$\text{gap-free}(\langle M_Q, \rightarrowtail_Q \rangle) \stackrel{\text{def}}{=} \forall p \in P,\ m, m' \in M_p^\Sigma : \big(\text{sub}(m) \to \text{sub}(m') \wedge\ m' \in M_Q\big) \Longrightarrow m \in M_Q$$

[2] This does not mean that a process may not submit the same "string of bits" twice. Formally, messages are opaque atoms, so any two submission events yield two distinct messages. An implementation can use message IDs to this effect.

[3] $\text{Dlv}(m,p)$ is a set, since an (incorrect) protocol may deliver a message m multiple times to some process p.

submit-ordered(Q). *An m-seq $Q = \langle M_Q, \rightarrowtail_Q \rangle$ is* submit-ordered *iff $\rightarrowtail_Q$ is an extension of the causal order relation $\rightarrow$ over the submit events of $M_Q \cap M^\Sigma$. Formally,*

$$\text{submit-ordered}(\langle M_Q, \rightarrowtail_Q \rangle) \stackrel{\text{def}}{=} \forall m, m' \in \left(M_Q \cap M^\Sigma\right) : \left(\text{sub}(m) \rightarrow \text{sub}(m')\right) \Longrightarrow \left(m \rightarrowtail_Q m'\right)$$

delivery-ordered(Q). *An m-seq $Q = \langle M_Q, \rightarrowtail_Q \rangle$ is* delivery-ordered *iff $\rightarrowtail_Q$ is an extension of the causal order relation $\rightarrow$ over the delivery events of $M_Q \cap M^\Delta$. Formally,*

$$\text{delivery-ordered}(\langle M_Q, \rightarrowtail_Q \rangle) \stackrel{\text{def}}{=} \forall m, m' \in \left(M_Q \cap M^\Delta\right), e \in \text{Dlv}(m), e' \in \text{Dlv}(m') : \left(e \rightarrow e'\right) \Rightarrow \left(m \rightarrowtail_Q m'\right)$$

terminated(Q). *An m-seq $Q = \langle M_Q, \rightarrowtail_Q \rangle$ is* terminated *iff every message m submitted by a correct process p is in M_Q. Formally,*

$$\text{terminated}(\langle M_Q, \rightarrowtail_Q \rangle) \stackrel{\text{def}}{=} \forall p \in P,\ m \in M_p^\Sigma : \text{correct}(p) \Longrightarrow m \in M_Q$$

3.3 Correctness Properties

Based on the previous definitions, we can state the following correctness property.

Correct Routing. *There exists a valid, minimal, complete, gap-free, submit-ordered, delivery-ordered, terminated m-seq Q.*

A protocol that transfers messages between processes in a manner that Correct Routing holds is considered *correct*. Formally speaking, Correct Routing does not require the physical existence of any m-seq. Rather, it is sufficient to show that one can, in theory, construct an m-seq where Correct Routing holds, given any history of events a protocol permits. That being said, our algorithmic solution, E-Cast, does indeed construct such an m-seq at runtime.

Correct Routing requires that any message m is delivered, in the right order, to the processes which the application considers destinations of m. Correct Routing is hard to split into independent properties, because it has to be ensured that all predicates (minimality etc.) hold for a *single* m-seq. Requiring that properties hold only for *delivered* messages (as traditionally done) is insufficient, if one wants to allow for messages whose destination set is empty, but which nevertheless influence the destination set of subsequent messages.

The following (variations of) familiar properties can easily be shown to follow from Correct Routing.

Validity. *No message is delivered that has not been submitted.*

Uniform Causal Order Delivery. *If the submission of $m \in M^\Sigma$ causally precedes the submission of $m' \in M^\Sigma$ then no process $p \in P$ delivers m' before m.*

Uniform Total Order Delivery. *For any pair of messages $m, m' \in M^\Delta$, if some process $p \in P$ delivers both m and m', and delivers m before m', then any process $p' \in P$ that delivers m and m' delivers m before m'.*

Uniform Agreement. *Any message m delivered by any process is delivered by every correct process in the destination set of m, in accordance with some m-seq Q.*

Termination. *Any message m submitted by a correct process is delivered to every correct process in the destination set of m, in accordance with some m-seq Q.*

If $\text{destset}(Q, m) \stackrel{\text{def}}{=} P$ for any Q and m, then the problem statement is equivalent to uniform atomic broadcast with causal order delivery [15]. Correct Routing is thus a generalization of uniform atomic broadcast, and all the known impossibility results apply.

4 Protocol

This section is structured as follows. First, the system model defined in Section 3.1 is extended with a concrete network and failure model. Then, E-Cast is specified in pseudo-code.

4.1 Network and Failure Model

Processes execute asynchronously, and communicate via messages passed over asynchronous, quasi-reliable FIFO channels; that is, if a correct process p sends a message m to a correct process p', then p' eventually receives m [1]. Processes may crash silently (not fail-stop), but otherwise behave as specified (no Byzantine failures).

We assume that processes have access to inaccurate but complete failure detectors. That is, a process may suspect another, correct process of being faulty, but every faulty process is eventually suspected of being faulty by every correct process. It is known that eventual weak accuracy (i.e., eventually *not* suspecting a correct process of being faulty) is required for liveness of uniform atomic broadcast and by extension E-Cast under the given system model [10]. As is common practice in systems involving consensus, we assume that the failure detectors are accurate enough to ensure liveness *in practice*.

Processes propagate their failure suspicions by submitting messages. A process p is defined *faulty* iff p crashes, or a process p' submits a message declaring p faulty[4]. This represents a form of process-controlled crash [15]. The existence of an application-specific function $\text{suspects}(m)$ is assumed, which maps every message m to a set of processes $\text{suspects}(m)$, which are declared permanently faulty by m. We define $\text{suspects}(\langle M_Q, \rightarrowtail_Q \rangle) \stackrel{\text{def}}{=} \bigcup_{m \in M_Q} (\text{suspects}(m))$ as a shorthand for the union of suspects over an m-seq Q. It is assumed that if a correct process p suspects another process p' of being faulty, p eventually submits a message m such that $p' \in \text{suspects}(m)$.

4.2 Protocol Idea

E-Cast is specified in terms of two roles: application and router. Application processes $A \subseteq P$ submit and deliver messages. Router processes $R \subseteq P$ do not submit or deliver messages; they act as message sequencers and intermediaries between application processes. It is permissible for a single process $p \in P$ to be both an application and a router process.

Assume routers agree on some m-seq Q, and assume that application processes receive messages of the form $\langle m,t \rangle$, where $m \in M_Q$, and $t \in \mathbb{N}_0$ is the index of the message m in Q. Further assume that $M_Q \subseteq M^\Sigma$. If every application process delivers messages in order of t, then Q is valid and delivery-ordered. If the order of t corresponds to the causal order of submission events, then Q is also send-ordered.

Assume routers send every message $m \in M_Q$ exactly to the processes in $\text{destsetpfx}(Q,m)$. If every correct process in the destination set of a message $m \in M_Q$ according to Q eventually receives m, and eventually delivers every unique message it receives, then Q is minimal, complete, and terminated. If Q is also gap-free, then Correct Routing holds. A full proof of correctness along these lines is provided in the appendix.

4.3 Consensus and Router Replication

The routers establish consensus on a set of messages that are delivered, and a total order of delivery for these messages. To this end, they *propose* and *learn* messages using uniform atomic broadcast with FIFO

[4] A single message submitted by a single process can declare a process faulty, but an application may still run arbitrarily elaborate monitoring and distributed agreement protocols before allowing a process to submit such a message.

Algorithm 1: Application Process

```
state variables
    t_max ← -1 ;                                   /* greatest timestamp of any delivered message */
    W ← ⟨⟩ ;                                       /* send window (queue of submitted, unacknowledged messages) */
upon submit m do
    append m to W
upon receive ⟨"stable",id(m)⟩ do
    remove m from W
    acknowledge m to user
upon receive ⟨"deliver",m,t⟩ from r do
    if t > t_max then                              /* if timestamp t of message m is greater than any previous timestamp */
        deliver m ;                                /* deliver m and remember the timestamp */
        t_max ← t
    send ⟨"confirm",id(m)⟩ to r ;                  /* in any case, confirm the message to the router which sent it */
periodically
    choose some router r
    foreach m ∈ W do
        send ⟨"route",m⟩ to r ;                    /* send all messages in the send window to some router, in submission order */
```

delivery order [15]. Every router maintains a growing m-seq of learned messages, which determines the messages to deliver and their order.

Depending on the implementation, the set of routers may change over time. The resulting problem is well known and corresponds to the reconfigurable state-machine problem [13, 24] or uniform atomic broadcast over a single dynamic group [7, 34]. We have implemented our own uniform atomic broadcast protocol for this purpose, but the details are outside the scope of this document.

4.4 Application Process

Pseudo-code for application processes is given in Algorithm 1. Submitted messages are sent in submission order, and received messages are delivered in timestamp order. When a message $\langle$"stable",$m\rangle$ is received, m is declared stable (i.e., uniformly delivered) and is removed from the send window. All unstable messages are periodically sent to some router. Any router is safe, though liveness requires that application processes eventually choose a correct router.

Some obvious optimizations have been left out for simplicity. Notably, assuming quasi-reliable FIFO channels, an application process a need not re-send any message m to a router r, which a had sent to r before. To avoid sending messages multiple times, an application process should only choose a new router if it learns that its current router has failed.

4.5 Router Process

Pseudo-code for router processes is given in Algorithm 2. Whenever a router *receives* a message that is not known to be stable, it *proposes* it by atomically broadcasting it to all routers, including itself. Whenever a router *learns* a new message m, it appends m to its m-seq Q and send window W. The lead router (see below), periodically sends all messages in its send window to their respective destination sets.

When all application processes in the destination set of a message m have confirmed m or have become suspects, m is proposed stable via atomic broadcast. When a router learns that a message m is stable, it can garbage-collect all state related to m. To later identify m as stable, an implementation may use session and sequence numbers.

For efficiency, only one router (the *leader*) should forward each message to its respective destination set. Given an atomic broadcast protocol between routers, leader election is easy to implement. However,

Algorithm 2: Router Process

```
state variables
    Q ← ⟨⟩ ;                                   /* learned m-seq */
    W ← ⟨⟩ ;                                   /* send window (queue of learned, unstable messages) */
    C ← ⟨⟩ ;                                   /* mapping of messages to destination processes that need to confirm */
upon receive ⟨"route", m⟩ do
    if m not stable then propose ⟨"route", m⟩ to all routers
    else send ⟨"stable", id(m)⟩ to submitter(m)
upon receive ⟨"confirm", id(m)⟩ from a do
    if C[m] exists then C[m] ← C[m]\{a}
upon learn ⟨"route", m⟩ from some router do
    if m not in Q then append m to Q and W
upon learn ⟨"stable", id(m)⟩ from some router do
    remove m from W
    remember m as stable
periodically if self is leader then
    foreach m ∈ W do
        C[m] ← destsetpfx(Q,m) \ suspects(Q)
        t ← the index of m in Q
        foreach a ∈ C[m] do send ⟨"deliver", m, t⟩ to a
periodically foreach m where C[m] exists do
    C[m] ← C[m] \ suspects(Q)
    if C[m] = {} then
        propose ⟨"stable", id(m)⟩ to all routers
        destroy C[m]
```

leader election is purely an optimization. Any router may safely consider itself a leader at any time.

Again, some optimizations have been left out. For instance, a router need not re-propose a message m which is already in Q when m is received. Also, it is inefficient to atomically broadcast "stable"-messages by themselves. An implementation may piggyback them onto "route"-messages. Finally, to avoid application processes having to repeat "route"-messages in order to receive a "stable"-message, a router process can safely send a message $\langle \text{"stable"}, \text{id}(m) \rangle$ to the application process that submitted m, as soon as the router process learns $\langle \text{"stable"}, \text{id}(m) \rangle$. For efficiency, an implementation should let a designated router process do this (e.g., the leader).

4.6 Quiescence

In addition to Correct Routing, E-Cast satisfies the following liveness property.

Quiescence. *For any message $m \in M_a^{\Sigma}$ submitted by a correct application process $a \in A$, eventually the application process receives a message $\langle \text{"stable"}, \text{id}(m) \rangle$.*

Quiescence ensures that messages can eventually be acknowledged to the user and garbage collected. A proof is provided in the appendix.

4.7 Implementation Notes

In practice, not every message changes the destination set of the messages that follow. So while destset is defined over m-seqs, an efficient implementation could maintain a (much shorter) sequence of configurations induced by the m-seq, and compute destset based on these configurations. We discuss this in more detail in Section 5.4.

Regarding memory usage, as Lamport [21] explains, it is possible to achieve consensus on an infinite m-seq Q with finite memory, through the use of checkpoints. However, garbage collection could never occur if router processes had to route any—arbitrarily old—message they received.

Fortunately, only *unstable* messages need to be routed. At the point where a router process $r \in R$ is informed that a message $m \in M$ is stable, r can safely garbage-collect all state related to m. If r later receives m from an application process $a \in A$, and r can tell that m is a stable message (easily solved by sequence numbers), it immediately responds $\langle$"stable", id$(m)\rangle$ to a.

With respect to latency, the protocol requires 5 message delays for delivering a submitted message in the common case. 1 for sending the submitted message to a router process, 3 for uniform consensus or atomic broadcast [11, 22], and 1 for sending the message to a destination process. This may seem much, but there are no restrictions on the pipelining of messages other than those imposed by limited buffer memory or the atomic broadcast protocol. The latter can be implemented in a way that pipelines messages, as explained by [33].

Finally, note that the application is not "in the loop". Messages are delivered asynchronously. Any extensive computation (e.g. query processing) or "state shipping" (e.g. database re-partitioning) that is performed by the application in response to a delivered message can be performed *outside* of E-Cast.

5 Partial Replication

To demonstrate that E-Cast is a suitable protocol for implementing elastic databases, one has to show that it provides a solution to partial replication and asynchronous reconfiguration. To this end, we first give a formal model for a partially replicated database. Based on this model, we make a formal problem statement. Then, we present a reduction of partial replication with strong consistency to E-Cast. Finally, we discuss how an implementation of partial replication based on E-Cast can avoid broadcast and perform asynchronous reconfiguration.

5.1 Application Model

The application consists of a set of client processes C and storage processes S. For each client process $c \in C$, a set of updates U_c, and a strict total order relation $\lhd_c$ on U_c exists, called the write order relation of c. It holds that $\forall c, c' \in C : c \neq c' \implies (U_c \cap U_{c'} = \{\})$.

Every storage process maintains a table. Tables are atoms whose exact nature is irrelevant. Let $\mathbb{T}$ denote the universe of tables. For each storage process $s \in S$, an initial table T_s^0, a set of updates U_s, and a strict total order relation $\blacktriangleleft_s$ on U_s exists. We call $\blacktriangleleft_s$ the execution order relation of s.

Definition 1 (Update)**.** *Let $U \stackrel{\text{def}}{=} \bigcup_{c \in C}(U_c)$ denote the set of all updates. Every update $u \in U$ is a function $u\colon \mathbb{T} \to \mathbb{T}$. Let $\mathbb{W} \stackrel{\text{def}}{=} \text{seq}(U)$ be defined as the set of all possible sequences (permutations) of updates in U. Assume $W = \langle u^0, u^1, ..., u^k \rangle$ is some sequence of updates of length k. We write $W(T)$ as a shorthand for the successive application of every update in $W \in \mathbb{W}$ on the table $T \in \mathbb{T}$, i.e., $W(T) \equiv u^k(\ldots u^1(u^0(T)))$.*

For every storage process $s \in S$, the execution order $\blacktriangleleft_s$ induces a sequence of updates $\langle U_s, \blacktriangleleft_s \rangle = \langle u_s^0, u_s^1, ..., u_s^k \rangle$. Executing the sequence of updates on T_s^0 induces a sequence of tables $\langle T_s^0, T_s^1 \ldots, T_s^{k+1} \rangle$. We define $\mathbb{T}_s \stackrel{\text{def}}{=} \{T_s^0, T_s^1 \ldots, T_s^{k+1}\}$ as the set of tables that a storage process s holds over its lifetime.

Definition 2 (Configuration)**.** *A configuration $\gamma\colon S \times U$ is a predicate over storage process, update pairs. Let Γ denote the universe of configurations. Every update $u \in U$ is also a higher-order function $u\colon \Gamma \to \Gamma$. Assume $W = \langle u^0, u^1, ..., u^k \rangle$ is some sequence of updates of length k. We write $W(\gamma)$ as a shorthand for the successive application of every update in $W \in \mathbb{W}$ on the configuration $\gamma \in \Gamma$, i.e., $W(\gamma) \equiv u^k(\ldots u^1(u^0(\gamma)))$.*

Define $T_s^{W,u} \stackrel{\text{def}}{=} \text{prefix}(W,u)(T_s^0)$ and $\gamma^{W,u} \stackrel{\text{def}}{=} \text{prefix}(W,u)(\gamma^0)$. The following is assumed.

Assumption 1 (Complete Configuration)**.** *If applying an update $u \in U$ to a table $T_s^{W,u} \in \mathbb{T}_s$ yields a different table, then $\gamma^{W,u}(s)$ must hold. Formally,*

$$\forall W = \langle U_W, \rightarrowtail_W \rangle \in \mathbb{W},\ s \in S,\ u \in U_W : T_s^{W,u} \neq u(T_s^{W,u}) \Longrightarrow \gamma^{W,u}(s)$$

In a sense, configurations are imperfect oracles. A configuration may not reliably predict whether an update has an effect on a particular table, but if it predicts that an update will *not* have an effect, then this is invariably true. In Section 5.4, we discuss possible implementations of configurations.

5.2 Problem Statement

Integrity. *No table of any storage process $s \in S$ is ever in a state that cannot be reached by applying some sequence of updates in U to T_s^0. Formally,*

$$\forall s \in S,\ T \in \mathbb{T}_s : \exists U' \subseteq U,\ W \in \text{seq}(U') : T = W(T_s^0)$$

Strong Consistency. *There exists a strict total order relation over U, call it $<$, which is an extension of the write order of every client process, and the execution order of every storage process. Formally,*

$$\forall u,u' \in U : \big(\big(\exists c \in C : u,u' \in U_c \wedge u \lhd_c u'\big) \vee \big(\exists s \in S : u,u' \in U_s \wedge u \blacktriangleleft_s u'\big)\big) \Longrightarrow u < u'$$

Atomicity. *If one storage process $s \in S$ executes an update $u \in U$, eventually the table of every correct storage process $s' \in S$ is in a state that can be reached by applying a sequence of updates to $T_{s'}^0$ which includes u. Formally,*

$$\forall s \in S,\ u \in U_s,\ s' \in S : \text{correct}(s') \Longrightarrow \exists T \in \mathbb{T}_{s'},\ U' \subseteq U_{s'},\ W \in \text{seq}(U') : T = u(W(T_{s'}^0))$$

Durability. *For any update $u \in U_c$ of a correct client process $c \in C$, eventually the table of every correct storage process $s \in S$ is in a state that can be reached by applying a sequence of updates to T_s^0 which includes u. Formally,*

$$\forall c \in C,\ u \in U_c,\ s \in S : \text{correct}(c) \wedge \text{correct}(s) \Longrightarrow \exists T \in \mathbb{T}_s,\ U' \subseteq U_s,\ W \in \text{seq}(U') : T = u(W(T_s^0))$$

An algorithm that executes updates of client processes at storage processes in a way that satisfies Integrity, Strong Consistency, and Atomicity is *safe*. An algorithm that satisfies Durability is *live*. An algorithm that is safe and live is *correct*.

Note that none of these properties make any reference to configurations. This is intentional. Configurations possibly allow an algorithm to *avoid* executing updates that have no effect. Executing an update that has no effect is inefficient, but is also safe, since it does not change the application state.

5.3 Reduction to E-Cast

The reduction to E-Cast is straight-forward. Every client or storage process is an E-Cast application process. Every update is a message. Assume a client process $c \in C$ submits each update $u \in U_c$ exactly once and in write order. Then for every client process $c \in C$, the write order $\lhd_c$ corresponds to the submission order over M_s^Σ. Further assume every storage process $s \in S$ executes each delivered update in delivery order. Then the execution order $\blacktriangleleft_s$ corresponds to the delivery order over M_s^Δ. Under this reduction, a protocol that is a safe solution of E-Cast is also safe solution of partial replication.

Proof of Integrity. Validity of E-Cast implies that $\forall s \in S : U_s \subseteq U$. Integrity follows immediately. □

Proof of Strong Consistency. Choose $<$ to be the submission and delivery event order $\rightarrow$. Strong Consistency follows from Uniform Total Order Delivery and Uniform Causal Order Delivery of E-Cast. □

If every update $u \in U$ is delivered to, and consequently executed by, every correct storage process $s \in S$, Atomicity and Durability hold trivially. However, this solution does not scale and is therefore not *elastic*. We now describe a better solution which takes advantage of configurations.

Note that if $\gamma^{W,u}(s)$ does not hold, then Atomicity and Durability hold trivially for s with respect to W and u, because the table $T_s^{W,u}$ held by s is already in a state as-if u had been applied. This insight leads to a safe definition of the function destset, which avoids broadcast.

Definition 3 (destset)**.** $\text{destset}(W,u) \stackrel{\text{def}}{=} \{s \in S \mid W(\gamma^0)(s)\}$

Corollary 1 (Complete Delivery)**.** *For any sequence of updates W, if $u(T_s^{W,u})$ yields a different table than $T_s^{W,u}$, then s is in the destination set of u according to W. Formally,*

$$\forall W = \langle W, \rightarrowtail_W \rangle \in \mathbb{W},\ s \in S,\ u \in (U_s \cap U_W) : u(T_s^{W,u}) \neq T_s^{W,u} \Longrightarrow s \in \text{destsetpfx}(W,u)$$

Proof of Atomicity. Atomicity follows immediately from Uniform Agreement of E-Cast. □

Proof of Durability. Durability follows immediately from Termination of E-Cast. □

5.4 Configuration and Reconfiguration

The reduction defines the function destset based on configurations. *Any* implementation where complete configuration (Assumption 1) holds is correct. But for scalability, one should avoid delivering updates that have no effect. Let us call a destset function *perfect* iff it holds that

$$\forall W = \langle W, \rightarrowtail_W \rangle \in \mathbb{W},\ s \in S,\ u \in (U_s \cap U_W) : u(T_s^{W,u}) \neq T_s^{W,u} \Longleftrightarrow s \in \text{destsetpfx}(W,u)$$

Note the double-ended arrow. While appealing in theory, a perfect destset function is impractical. For example, to decide whether a process s should be in the destination set of an update u for an object with key x, it is necessary to know, with certainty, whether the table stored in s contains an object with key x at the point where u is delivered. Decisions like this require a lot of configuration state, possibly the table itself. There is obviously a trade-off between the amount of configuration state and the number of updates that are delivered unnecessarily.

An important contribution of E-Cast is its clear separation of concerns regarding data placement and the order of update execution. This simplifies modeling and protocol design, and makes it possible to implement arbitrary trade-offs between configuration state and message overhead without compromising consistency.

As a concrete example, assume that a table is a set of data objects, each with a unique key. Data objects are distributed across storage processes according to some replicated, consistent hashing scheme, say successor-list replication [37]. When a new storage process s seeks to join the system, it sends a special join update u. The destination set of this update are the processes S', which, according to configuration $\gamma^{W,u}$, may contain data objects belonging to s according to the next configuration $u(\gamma^{W,u})$. Every storage process $s' \in S'$ holds the table $T_{s'}^{W,u}$ at the point where u is delivered, and $u(T_s^{W,u}) \subseteq \cup_{s' \in S'}(T_{s'}^{W,u})$ by definition of S'. Thus, it should be easy to construct the table $u(T_s^{W,u})$.

Removing a storage process s works analogously. The only difference is that s may not be the process which submits the removal update u for s. If s is faulty, some other, correct process $p' \neq p$ will submit u. Regardless of whether u is a process join or remove update, at the point where u is delivered, there is already system-wide *consensus* on the next configuration $u(\gamma^{W,u})$. Thus, shipping application state (e.g. table snapshots) can be done *asynchronously*, outside E-Cast, which enables elasticity.

6 Comparison to State Machines and Group Communication

In the state-machine approach, fault-tolerance is provided by replicating a state-machine across a set of processes, which—assuming a common initial state—reduces to the problem of consensus on a sequence of commands [35]. Various models and protocols for reconfigurable state machines, i.e., ways of changing the set of replicas, have been proposed [13, 23, 24]. All are based on the idea of consensus on an interleaving of the sequence of regular commands with a sequence of reconfiguration commands.

In comparison, group communication is based on consensus on a sequence of views (which are roughly equivalent to configurations), but does not necessarily require consensus on a total order of messages. By giving different ordering and delivery guarantees on messages with respect to views, a wide spectrum of consistency guarantees is established [12].

In comparison, E-Cast *does* require consensus on a total order of message delivery. The delivery order determines the destination processes of each message. There is no separate protocol for reconfiguration and membership (at least for application processes), because configuration and membership are expressed through the function destset and hence determined solely by the order of messages.

To allow for a better comparison of the three approaches, let us define system state as a pair of (global) configuration state, and the (local) application state of every process. We observe that all three approaches—reconfigurable state machines, group communication, and E-Cast—establish consensus on a partial order of such system states. In reconfigurable state machines, there is consensus on a total order of system states, and every process[5] holds the complete system state. In group communication, there is consensus on a total order of configuration state, and some protocol-specific partial order of application state. In E-Cast, there is consensus on a total order of system state, as in the state-machine approach. But in contrast to the state-machine approach, E-Cast does not imply full replication of the application state.

E-Cast can therefore be interpreted as a generalization of reconfigurable state machines, which separates the local, application state of processes from the global, configuration state that is necessary for consistency. As we have demonstrated, the model is almost as simple to specify, use, and implement as state-machine replication, but also as general and efficient as total order multicast over arbitrary groups, because it enables applications that avoid any unnecessary message broadcast and state redundancy.

7 Conclusions

This technical report presented a novel generalization of total order multicast, E-Cast, designed to support strongly consistent partial replication in cloud environments. The key insight behind E-Cast is to enforce consensus on the history of the messages rather than separately on the order of individual messages and membership. For this purpose, E-Cast introduces a new important concept, the destination processes of each message as an application-specific function of the prefix of the message according to (an extension of) the total order of delivery. By doing so, the system becomes elastic: reconfigurations due to changes in membership do not interrupt the flow of messages and it is possible to support a large number of arbitrarily overlapping groups in an intellectually manageable manner.

References

[1] M. K. Aguilera et. al. Heartbeat: A timeout-free failure detector for quiescent reliable communication. In *Proc. WDAG '97*, 1997.

[2] Y. Amir and C. Tutu. From total order to database replication. In *Proc. ICDCS '02*, 2002.

[3] Y. Amir et. al. The totem single-ring ordering and membership protocol. *ACM TOCS*, 13, 1995.

[5] By "process", we mean any process which actually contains application logic, i.e., the state-machine implementation.

[4] Y. Amir et. al. A low latency, loss tolerant architecture and protocol for wide area group communication. In *Proc. ICDSN '00*, 2000.

[5] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, and M. Zaharia. Above the clouds: A berkeley view of cloud computing. Technical report, UC Berkeley RAD Labs, 2009.

[6] K. Birman. A history of the virtual synchrony replication model. In *Replication: Theory and Practice*. Springer, 2010.

[7] K. Birman, D. Malkhi, and R. V. Renesse. Virtually synchronous methodology for building dynamic reliable services, 2010.

[8] K. P. Birman. The process group approach to reliable distributed computing. *CACM*, 36(12), 1993.

[9] K. P. Birman and R. V. Renesse. *Reliable Distributed Computing with the ISIS Toolkit*. IEEE Computer Society Press, 1994.

[10] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *JACM*, 43(2), 1996.

[11] B. Charron-Bost and A. Schiper. Uniform consensus is harder than consensus. *J. of Algorithms*, 51(1), 2004.

[12] G. V. Chockler et. al. Group communication specifications: A comprehensive study. *ACM Comp. Surv.*, 33(4), 2001.

[13] G. V. Chockler et. al. Reconfigurable distributed storage for dynamic networks. *JPDC*, 69(1), 2009.

[14] S. Das et. al. G-store: a scalable data store for transactional multi key access in the cloud. In *Proc. SoCC '10*, 2010.

[15] X. Défago et. al. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comp. Surv.*, 36(4), 2004.

[16] B. B. Dept and B. Ban. Adding group communication to java in a non-intrusive way using the ensemble toolkit. Technical report, Cornell University, 1997.

[17] D. Dolev and D. Malki. The transis approach to high availability cluster communication. *CACM*, 39, 1996.

[18] U. Fritzke Jr. and P. Ingels. Transactions on partially replicated data based on reliable and atomic multicasts. *Proc. ICDCS'01*, 0, 2001.

[19] B. Kemme and G. Alonso. Don't be lazy, be consistent: Postgres-r, a new way to implement database replication. In *Proc. VLDB '00*, 2000.

[20] B. Kemme, A. Bartoli, and O. Babaoglu. Online reconfiguration in replicated databases based on group communication. *Proc. DSN '01*, 0, 2001.

[21] L. Lamport. Generalized consensus and paxos. Technical Report MSR-TR-2005-33, Microsoft Research, 2005.

[22] L. Lamport. Lower bounds for asynchronous consensus. *Distributed Computing*, 19(2), 2006.

[23] L. Lamport, D. Malkhi, and L. Zhou. Stoppable paxos. Technical report, Microsoft Research, 2008.

[24] L. Lamport, D. Malkhi, and L. Zhou. Reconfiguring a state machine. *SIGACT News*, 41(1), 2010.

[25] C.-H. Lee, J.-D. Lee, and H.-Y. Bae. Dpr: A dynamic partial replication protocol based on group communication for a web-enable database cluster. In *Proc. APWeb'03*, 2003.

[26] A. A. Lima, C. Furtado, P. Valduriez, and M. Mattoso. Parallel olap query processing in database clusters with data replication. *Distrib. Parallel Databases*, 25(1-2), 2009.

[27] K. Ostrowski, K. Birman, and D. Dolev. Quicksilver scalable multicast (qsm). In *Proc. NCA '08*, 2008.

[28] K. Ostrowski, K. Birman, D. Dolev, and J. H. Ahnn. Programming with live distributed objects. In *Proc. ECOOP '08*, 2008.

[29] F. Pedone, R. Guerraoui, and A. Schiper. The database state machine approach. *Distrib. Parallel Databases*, 14(1), 2003.

[30] F. Pedone and A. Schiper. Modular approach to replication for availability. In *Replication: Theory and Practice*. Springer, 2010.

[31] C. Plattner, G. Alonso, and M. T. Özsu. Extending dbmss with satellite databases. *VLDBJ*, 17(4), 2008.

[32] V. Ramasubramanian, T. L. Rodeheffer, D. B. Terry, M. Walraed-Sullivan, T. Wobber, C. C. Marshall, and A. Vahdat. Cimbiosys: A platform for content-based partial replication. In *Proc. NSDI'09*, 2009.

[33] B. Reed and F. P. Junqueira. A simple totally ordered broadcast protocol. In *Proc. LADIS '08*, 2008.

[34] A. Schiper. Dynamic group communication. *Distributed Computing*, 18(5), 2006.

[35] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comp. Surv.*, 22(4), 1990.

[36] A. Sousa, R. Oliveira, F. Moura, and F. Pedone. Partial replication in the database state machine. In *Proc. NCA '01*, 2001.

[37] I. Stoica et. al. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proc. SIGCOMM '01*, 2001.

[38] R. van Renesse, K. P. Birman, and S. Maffeis. Horus: a flexible group communication system. *CACM*, 39, 1996.

[39] M. Wiesmann and A. Schiper. Comparison of database replication techniques based on total order broadcast. *IEEE Transactions on Knowledge and Data Engineering*, 17(4), 2005.

A Proofs

A.1 Correctness

Theorem 1. *E-Cast is correct.*

As discussed in Section 4.5, router processes propose and learn messages via uniform atomic broadcast with FIFO delivery. It is easy to see that, by definition of uniform atomic broadcast, all router processes agree on an ever-growing m-seq Q. To proof that Correct Routing holds, we proof that this m-seq Q is valid, minimal, complete, gap-free, submit-ordered, delivery-ordered, and terminated.

Lemma 1. *Q is valid.*

Proof. To appear in Q, a message m must be learned by some router r. To be learned by r, m must previously be proposed by some router r'. For m to be proposed by r', r' must previously receive a message $\langle$"route",$m\rangle$. Assuming no Byzantine failures, an application process a must previously send the message $\langle$"route",$m\rangle$ to r', which in turn implies that m must previously be submitted. □

Lemma 2. *Q is minimal.*

Proof. No application process ever delivers a message m for which it did not receive a $\langle$"deliver",$m,t\rangle$ message. No router process ever sends a message $\langle$"deliver",$m,t\rangle$ to any application process that is not in destsetpfx(Q,m). Minimality of Q follows immediately. □

Lemma 3. *Q is submit-ordered.*

Proof. It is to show that for any pair of messages $m,m' \in M_Q$ where sub$(m) \rightarrow$ sub(m'), it holds that $m \rightarrowtail_Q m'$. We first proof that this holds if $m,m' \in M_a^\Sigma$ for some application process a (Case 1, FIFO order delivery). We then proof the general case (Case 2, causal order delivery).

Case 1: Assume $m,m' \in M_a^\Sigma$ for some application process a. For the message m' to be learned by any router process, it must first have been proposed by some router process, call it r. To be proposed by r, r must have received a message $\langle$"route",$m'\rangle$ from a. At that point, either m was already learned or not. If m was already learned, then m must be in Q at that point, so clearly $m \rightarrowtail_Q m'$. If m was not already learned, then m can also not have been acknowledged. As obvious from the application process algorithm, and since processes communicate through quasi-reliable FIFO channels, r must have received $\langle$"route",$m\rangle$ before it received $\langle$"route",$m'\rangle$. Therefore, r must have proposed m before m'. Since the atomic broadcast protocol guarantees FIFO order delivery, every router must learn m before m'. It follows that $m \rightarrowtail_Q m'$.

Case 2: Assume m and m' were submitted by different application processes. Without restriction of generality, say m was submitted by application process a, and m' was submitted by application process a'. For $\text{sub}(m) \rightarrow \text{sub}(m')$ to hold, there must be a message m'' submitted by a, such that $\text{sub}(m) \rightarrow \text{sub}(m'')$ or $m = m''$, and m'' is delivered to at least one application process. (Otherwise there could be no chain of causality between $\text{sub}(m)$ and $\text{sub}(m')$.) But to be delivered to any application process, m'' must first be learned by some router. Therefore, m'' must precede m' in Q; i.e., $m'' \rightarrowtail_Q m'$. If $m = m''$, then obviously $m \rightarrowtail_Q m'$. If $m \neq m''$, then Case 1 applies and $m \rightarrowtail_Q m''$. By transitivity of $\rightarrowtail_Q$, it holds that $m \rightarrowtail_Q m'$. □

Lemma 4. *Q is delivery-ordered.*

Proof. It is to show that for any pair of messages $m, m' \in M_Q$, where the delivery of m, call the event e, precedes the delivery of m', call the event e'; i.e., $e \rightarrow e'$, it holds that $m \rightarrowtail_Q m'$. We first proof that this holds if $m, m' \in M_a^\Delta$ for some application process a (Case 1). We then proof the general case (Case 2).

Case 1: The timestamp t of any message $\langle$"deliver", $m, t\rangle$ sent by any router process, is the index (position) of m in Q. As obvious from the application process algorithm, application processes deliver messages strictly in timestamp order. It follows that for any application process a, for any pair of messages $m, m' \in M_a^\Delta$ where a delivers m before m', it holds that $m \rightarrowtail_Q m'$.

Case 2: For $e \rightarrow e'$ to hold, there must be a message m'' submitted by a, such that $e \rightarrow \text{sub}(m'') \rightarrow e'$. By analogous argument to the proof of submit-orderedness, it must hold that $m \rightarrowtail_Q m''$, and $m'' \rightarrowtail_Q m'$. By transitivity of $\rightarrowtail_Q$, it holds that $m \rightarrowtail_Q m'$. □

Lemma 5. *Q is gap-free.*

Proof. Choose any pair of messages m, m' submitted by an application process a, where $\text{sub}(m) \rightarrow \text{sub}(m')$, and m' is in M_Q. It is to show that m is also in M_Q. We assume that m is not in M_Q and proof by contradiction.

Application processes send "route"-messages strictly in submission order. Since m is not in M_Q, m cannot be acknowledged at any point, so for any router r to which a sends a message $\langle$"route", m'$\rangle$, a first sends a message $\langle$"route", m$\rangle$.

Processes communicate through quasi-reliable FIFO channels. Therefore, any router that receives a message $\langle$"route", m'$\rangle$ must first receive a message $\langle$"route", m$\rangle$. Since m is not in M_Q and thus obviously not stable, any router process that receives and proposes m' must also, previously, receive and propose m. Since the atomic broadcast algorithm preserves FIFO order, for m' to be learned, m must be learned first. Since m' is in M_Q, m must be in M_Q as well. Contradiction, hence proof. □

Lemma 6. *Q is complete.*

Proof. We proof by contradiction. Assume Q is *not* complete; i.e., there is some message $m \in M_Q$ that is not delivered by some correct application process $a \in \text{destsetpfx}(Q, m)$.

Assuming at least one correct router process that eventually considers itself a leader, every message $m \in M_Q$ is eventually sent to and received by every correct application process, including a. To be precise, a must receive a message $\langle$"deliver", $m, t\rangle$. Consider the first time a receives $\langle$"deliver", $m, t\rangle$. For a not to deliver m, a must have previously received a message $\langle$"deliver", $m', t'\rangle$ where $t' > t$.

Router processes send "deliver"-messages in order of Q, that is timestamp order. Since a had not received and confirmed m yet, and a is a correct process, the router process that sent $\langle$"deliver", $m', t'\rangle$ must previously have sent $\langle$"deliver", $m, t\rangle$. Processes communicate through quasi-reliable FIFO channels, so a cannot have received $\langle$"deliver", $m', t'\rangle$ before $\langle$"deliver", $m, t\rangle$. Contradiction, hence proof. □

Lemma 7. *Q is terminated.*

Proof. Assuming every correct application process eventually sends its submitted messages to a correct router process that eventually considers itself a leader, every submitted message is eventually learned by every correct router process. Q is terminated by definition. □

Proof of Theorem 1. Theorem 1 follows by definition from Lemmas 1 through 7. □

A.2 Quiescence

Theorem 2. *For any possible execution of E-Cast, Quiescence holds.*

Let Q be the ever-growing m-seq that the router processes agree on. We prove the following Lemma.

Lemma 8. *Every message $m \in M_Q$ is eventually considered stable by every correct router process.*

Proof. We assume there is at least one correct router process that eventually considers itself a leader. For any message $m \in M_Q$, eventually one such router process will send $\langle$"deliver", m, t$\rangle$ messages (with some irrelevant timestamp t) to all processes in $D = \text{destsetpfx}(Q, m)$. Let us refer to this router process as r.

Eventually all correct application processes will respond with $\langle$"confirm", id$(m)\rangle$ to r. Eventually r receives all the $\langle$"confirm", id$(m)\rangle$ messages sent by correct application processes. It is assumed that the failure detectors used by application processes are complete. Therefore, any faulty process $a' \in D$ which does not confirm m is eventually suspected of failure, i.e. a correct application process submits a message $m' \in M_Q$ such that $a' \in \text{suspects}(m')$.

At the point where r has received $\langle$"confirm", id$(m)\rangle$ from all correct application processes in D, and has learned of the failure of all faulty application processes in D, r proposes $\langle$"stable", id$(m)\rangle$ to all router processes. Eventually, all correct router processes learn this message and consider m stable. □

Proof of Theorem 2. Lemma 8 guarantees that every message m in M_Q is eventually considered stable by every correct router process. As obvious from the router process algorithm, from the point on where a router process considers a message m stable, it responds to $\langle$"route", $m\rangle$ messages with a message $\langle$"stable", id$(m)\rangle$. Since the application process a that submitted m is correct, it eventually re-sends the message $\langle$"route", $m\rangle$ to a correct router process and eventually receives a message $\langle$"stable", id$(m)\rangle$ in response. □