Asynchronous Concurrency Control

Tristan Schmelcher

Abstract

We present an "asynchronous" concurrency control system called ACC-1 and show that it can be used to substantially increase the performance of massively distributed systems without sacrificing consistency. Applications are given for real-time collaborative editing, distributed databases, distributed shared memory, and autoparallelization. The correctness of the central algorithms is proven formally using group theory. Fault tolerance is analyzed and a recovery procedure is presented.

Index Terms

Concurrency, distributed systems, distributed applications, distributed databases, distributed architectures, multiprocessors.

CONTENTS

I. Introduction

In networks of computers that read and write to replicated shared state, distributed concurrency control is needed so as to enforce consistency. Traditionally this is achieved via additional, synchronous communication that coordinates which computers have permission to access the replicated data at any given time—distributed locking is the typical mechanism. As such systems are scaled up, the network delay makes this synchronous communication increasingly be performance-limiting. As a result, massively distributed systems face a trade-off between performance and consistency.

We present a distributed concurrency control system that we call ACC-1 (Asynchronous Concurrency Controller number one). Unlike traditional such systems, ACC-1's communication is almost entirely asynchronous, in the sense that network nodes do not have to wait for responses from others before manipulating replicated shared state. Despite this, ACC-1 can provide full consistency for any distributed system. As a result, its use can enable systems to vastly reduce the need to trade-off consistency for performance.

Throughout this paper, many assumptions are made; some of them are thoroughly unreasonable. However, the purpose of these assumptions is not to restrict the paper to a subset of situations, but rather to allow for the analysis to be done in layers. By the end of the paper, many of the initial assumptions will have been eliminated.

II. Notation

All indices start at 1.

Let s, s_1, and s_2 be sequences in mathematics. Then,

$s[i]$ denotes the i^{th} element of s,

$\overline{s}$ denotes the sequence containing the contents of s in reverse order,

$s_{x..y}$ denotes the subsequence of s containing those elements with indices in the range $[x, y]$,

$\{e_1, e_2, ..., e_n\}$ denotes the literal sequence of the elements e_1, e_2, ..., e_n,

$s_1 + s_2$ denotes sequence concatenation,

$\#s$ denotes the length of s,

s_{-x} denotes $s_{(1+x)..\#s}$—i.e., s with the first x elements removed,
and $s_{\overline{-x}}$ denotes $\overline{s_{-x}}$—i.e., s with the last x elements removed.

The same applies for a list `L` in pseudo-code.

Denote the set of sequences over the elements of some set S as $[S]$.

III. Fundamental 2-Node System

This section introduces the fundamental ACC-1 system, which is used as a basis for those we present later. It only considers distributed systems that contain exactly two computers (hereafter "nodes") which are connected by exactly one communication channel.

A. Assumptions

- The shared state of the system is something resembling a database; it is innert data which is operated on by transactions that transform it from a previous state to a next state.[1]
- The communication channel is bi-directional, in-order, reliable, and asynchronous.[2]
- The shared state is fully replicated at both nodes.[3]
- All transactions operate *only* on the data in the database; they do not perform any I/O with the external world, nor do they store any information externally for future processing. Thus, reading a database replica's state with an external program is a separate kind of operation from a transaction and for the moment we will not consider it.[4]

We will model database state and transactions abstractly using a simple algebra. Expressions in the algebra will represent descriptions of transactions to perform, the set of which will be referred to as T.[5]

Define the *sequencing operator* $* : T \times T \to T$ as an operator such that for any two transactions A and B, $A * B$ is the transaction that atomically performs A and then B. Any real-world system will either already have such an operator or will be able to trivially have it added.

We will assume that T forms a group under $*$ (with I denoting the identity element and A^{-1} denoting the inverse of A). From the definition of a group [1], this is equivalent to assuming that, for all A, B, and C in T,

$$\begin{aligned} (A * B) * C &= A * (B * C) && Associativity \\ I * A &= A * I = I && Identity\ element \\ A * A^{-1} &= A^{-1} * A = I && Inverse\ element \end{aligned}$$

For real-world systems, the associativity property will clearly follow from the intuitive meaning of $*$.

Further, an identity element will also clearly either already exist or be trivial to add, since it corresponds to a simple no-op.

The inverse element assumption is less clearly reasonable. Many real-world transaction systems have operations that cannot be undone by a single inverse transaction. (For example, a simple "write 3 to X" is not invertible.) However, it should be possible to make most systems satisfy this by having the database back-up any state that is unrecoverably overwritten. A special transaction for resurrecting the backed-up state could then be used to undo transactions that otherwise would not be invertible.[6]

With these assumptions made, we can now proceed to define ACC-1 for the two node case.

[1] We do not assume anything about the structure of states or transactions, so this is likely to be a valid assumption for at least the vast majority of real systems.

[2] e.g., TCP/IP, at least statistically, in the absence of an attacker, and to the extent that connections are not lost.

[3] This is of course sometimes undesirable in reality, but we will address that later.

[4] We will later consider systems where it is necessary to unify these two types of operations.

[5] Note importantly that the elements of T correspond to transaction code in real systems, *not* a particular execution of such code. For example, "read X and write X+1 to Y" might be an element of T, but "read X = 3 and write X+1 = 4 to Y" is not, because it corresponds to a particular execution in which "X" was 3.

[6] We will see though that in practice such modification to the system is not always necessary.

B. Definition

Let T' be any set and let $\kappa : T' \to T$ and $\delta : (T' \times T') \to T'$ be any computable functions satisfying the identity that, for all $A' \in T'$ and $B' \in T'$,[7]

$$\kappa(B') * \kappa(\delta(A', B')) = \kappa(A') * \kappa(\delta(B', A')) \tag{1}$$

We will call this the *symmetry criterion*.

Note that since the group T, the set T', and the functions κ and δ have been defined abstractly, ACC-1 will not be a single system, but rather a template for making a concurrency control system for particular choices of those objects.

The purpose of those objects can be understood intuitively as follows.

The set T' is essentially just a set whose elements correspond to those in T, but potentially with extra information added. We will refer to the elements of T' as *extended transactions*.

The κ function simply maps elements of T' to those in T to which they correspond. It can be thought of as "cleaning" the object of its extra information (if any).

The δ function will be used by ACC-1 to perform conflict resolution. It operates on the transactions in T' and thus is able to use their extra information (if any) to help compute its result. (It is for this purpose that we allow for the possibility of such extra information.)

We will introduce various possible choices for these objects later. In some cases, δ will have no need for any extra information and so we will choose $T' = T$ and $\kappa(A) = A$. For now though we will retain this generality.

Further, to aid in the later re-use of this section's analysis, we will consider only the ACC-1 components of the system. At each node there shall be an ACC-1 component that communicates with the one in the other node. Together they are responsible for synchronizing the replicas as follows.

- Each node's ACC-1 component communicates with the local system and also with the remote node's ACC-1 component via the communication channel.
- Communication between the ACC-1 component and the rest of its local node is *synchronous*,[8] while communication between the two ACC-1 components is *asynchronous*.[9]
- Each message passed between ACC-1 and the local node encodes an instances of T'. Each message passed between the two ACC-1 components either encodes an instance of T' or is a simple acknowledgement message that contains no data.[10] We will denote the acknowledgement message as α.
- The implementation of the non-ACC-1 portion of each node must satisfy the properties that:
  1) Whenever it receives any $A' \in T'$ from its ACC-1 component, it applies $\kappa(A')$ to its local replica.
  2) Precisely whenever it applies any transaction $A \in T$ to its local replica for any reason other than rule 1, it sends some $A' \in T'$ with $\kappa(A') = A$ to its ACC-1 component.
  3) There is a local critical section object that must be held by the sender in the above two situations, and in each case the transaction's execution is done during the lock of that object in which the message is sent.

  These properties guarantee that the ACC-1 component will be able to track the changes in the local node's replica and make its own changes as needed.

[7] Actually, it only really needs to satisfy this for all the ordered pairs $(A', B') \in T' \times T'$ for which its result will actually be computed in practice. This distinction will later prove relevant.

[8] i.e., messages are handled by the recipient before the sender continues execution. These are the same semantics as those of a procedure call.

[9] i.e., the sender does not wait for an answer before continuing execution.

[10] These acknowledgement messages are *not* the same as any acknowledgements that an underlying transport layer, such as TCP/IP, may send (and they cannot be combined with them). Rather, they are part of the application's own protocol.

- Each ACC-1 component's internal state is a list of extended transactions. We call this list the "skew sequence" and denote it in the algorithm as `S`.
- Each ACC-1 component operates by responding to events, of which there are three kinds:
  1) Reception of some message $A' \in T'$ from the local node (called "transaction issue").
  2) Reception of α on the connection ("acknowledgement reception").
  3) Reception of some $A' \in T'$ on the connection ("transaction reception").
- The system is initialized with empty skew sequences and with some initial replica state at each node (which need not be the same, but typically would be).
- The ACC-1 components' pseudo-code for handling events is as follows.
  - Transaction issue[11]

```
let A' = the transaction received from the local
        node (synchronously) that has been executed
append A' to S
send A' to the remote node (asynchronously)
```

  - Acknowledgement reception

```
lock critical section
remove the first element from S
unlock critical section
```

  - Transaction reception

```
init A' := the transaction description received
lock critical section
send an acknowledgement to the remote node (asynchronously)
for i = 1 to #S
        let A'' = delta(A', S[i])
        let Si' = delta(S[i], A')
        set S[i] := Si'
        set A' := A''
end
send A' to the local node to be executed (synchronously)
unlock critical section
```

(Note that there are two events that are triggered by receiving messages and two events that trigger messages to be sent. One event—transaction reception—is of both types.)

Observe that none of the pseudo-code above performs any operation that would require a node to wait for a message from another. Therefore, the number of round-trips of network latency incurred in the running time of a process that issues transactions is *zero*. By contrast, in a traditional concurrency control system, issuing each transaction would require acquiring a distributed lock, which would incur up to one round-trip of network latency (though less on average because we might already have a lock). Therefore, in massively distributed systems where network latency is extremely high compared to computation time, such a process in ACC-1 will enjoy *greatly improved performance*.[12]

An additional potential benefit of ACC-1 is that the lack of distributed locking implies that there cannot be distributed deadlock.

Neither of the above will be of use if the system's results are not correct—the subject of the next section.

[11] We do not lock the critical section here because it is already locked by the message sender.

[12] Though when we later generalize to other kinds of systems to eliminate our assumptions, the potential performance gain will no longer be boundless.

C. Behaviour

Since the correctness of ACC-1's behaviour is not at all obvious (nor is what it even *means* for it to be correct), we will perform a rigorous analysis. First, a number of definitions are necessary.

Observe that since event handling in each ACC-1 component is atomic and is inseparable from any corresponding local transaction execution due to the synchronous local message passing, we can number events at a node with sequential integers such that each local event (and any corresponding transaction execution on the local replica) starts after the event with the previous number completes and completes before the event with the next number starts.

We will use e as our canonical event number variable, and we will refer to the event with number e as the e^{th} ("eeth") event. The 1^{st} event at some node is thus the first transaction issue or transaction/acknowledgement reception event that occurs there; the 2^{nd} is the one after, and so on. As a convention, we will also define the 0^{th} ("zeroth") event at a node as being the "event" of that node being initialized.

Definition 1: For either node N, $\neg N$ denotes the remote node.

Definition 2: $S(N, e)$ denotes N's skew sequence upon completing the e^{th} event there, and $L(N, e)$ is an abbreviation for $\#S(N, e)$.

Definition 3: $c_r(N, e)$ is the count of the number of receive events (i.e., transaction reception events and acknowledgement reception events) that occurred at N from after the time of initialization up to and including event number e.

Definition 4: $n_t(N, c)$ is the event number of the c^{th} transmit event (i.e., transaction issue event or transaction reception event) at N, or 0 if $c = 0$.

Definition 5: $k(N, e)$ is an abbreviation for $n_t(\neg N, c_r(N, e))$. Since every receive event is caused by a past transmit event at the other end, this is the event number of the last transmit event at $\neg N$ whose message was received at N by the time of event number e (inclusive) at N—or, intuitively, this is the number such that N "knows" after e about the events that happened at $\neg N$ up to (and including) event number $k(N, e)$.

Definition 6: The *differential state* $B(N, e)$ is the product of all transactions executed on the replica at node N from after the time of initialization up to and including the moment after event number e, in the order of their execution. Therefore this is a transaction that, if executed on the replica state that node N started with, would produce it's state at the moment after the e^{th} event there. (When combined with knowledge of the initial state, this functions as a definition of the replica state after event e.)

Definition 7: The *committed differential state* (or simply *committed state*) at some node N after some event number e is:

$$\begin{aligned} C(N, e) &= B(N, e) * \prod_{i=1}^{L(N,e)} \kappa(\overline{S(N, e)}[i])^{-1} \\ &= B(N, e) * \kappa(\overline{S(N, e)}[1])^{-1} * \kappa(\overline{S(N, e)}[2])^{-1} * ... * \kappa(\overline{S(N, e)}[L(N, e)])^{-1} \end{aligned}$$

Intuitively, this is the differential state that you would get if you undid all the transactions in N's skew sequence on its local replica at the moment after event number e.

Definition 8: T'_α denotes the set $T' \cup \{\alpha\}$; i.e., the set of extended transactions, but with the acknowledgement message included. This is the set of precisely those messages that are carried by the communication channel.

Definition 9: $F(N, e)$ is the sequence over T'_α of all messages that were sent by N (in order of transmission) up to the moment after the e^{th} event at N and that were not received by $\neg N$ up to the moment after event number $k(N, e)$ at $\neg N$.

Definition 10: $p(N, e, i)$ is the index of the i^{th} extended transaction in $F(N, e)$, or $\#F(N, e) + 1$ if there are less than i extended transactions.

Definition 11: $f_A : T' \times [T'] \to T'$ is the function such that, for all $A' \in T'$ and $S \in [T']$, $f_A(A', S)$ is the result that the transaction reception event handler would compute for the transaction to execute if it received A' with its skew sequence initially equal to S.

Definition 12: $g : [T'_\alpha] \times [T'] \to [T']$ is the function such that, for all $m \in T'_\alpha$ and $S \in [T']$, $g(m, S)$ is the final value of the skew sequence that a node would have if it received the messages in m (in order and with no intervening events) and with its skew sequence initially equal to S.

ACC-1's fundamental property can now be stated.

Theorem 1: For all N and e,

$$C(N, e) = B(\neg N, k(N, e)), \tag{2}$$

$$F(N, e) \; contains \; exactly \; L(N, e) \; transactions \; and \; L(\neg N, k(N, e)) \; acknowledgements, \tag{3}$$

and $\forall i \in [1, L(N, e)]$,

$$S(N, e)[i] = f_A(F(N, e)[p(N, e, i)], X) \tag{4}$$
$$where \; X = g(F(N, e)_{1..(p(N,e,i)-1)}, S(\neg N, k(N, e)))$$

(2) can be read intuitively as "the committed state is always equal to a past actual state of the opposite node (in particular, its state as of the last message from it)". This is the important part of the theorem.

(3) is self-explanatory and not of great interest.

As for (4), given the meanings of f_A and g, the right-hand side can be read as "the value that the opposite node would compute for the i^{th} oldest unacknowledged message if it were to arrive before any other events occur there". The equation itself can thus be read as "every element in the skew sequence is equal to the value that the opposite node would compute for the corresponding sent message".

A rigorous proof of Theorem 1 is long and not particularly enlightening, so we have relegated it to Appendix I.

D. Intuitions

It is illustrative now to consider what the implications of Theorem 1 are for our intuitions regarding ACC-1. As mentioned above, it is mostly just the first part of the theorem that is of importance, and it's meaning is that "the committed state is always equal to a past actual state of the opposite node (in particular, its state as of the last message from it)".

Recall that the committed state is defined as the actual state with the inverse of the skew sequence applied. Therefore, whenever the skew sequence is empty, the committed state equals the actual state.

Thus a special case of Theorem 1 is that whenever the skew sequence is empty, the local state is the past remote state as of the last message from it.

A further special case is that whenever the skew sequences at both ends are empty, their states are equal.

What this tells us intuitively is that—despite the lack of any distributed locking—nodes in ACC-1 are only ever transiently out-of-sync. The quiescent state of the system is for the two nodes to have the same state.

Note also that we can derive from this a meaning for the skew sequence: it is a sequence of transactions by which our replica's state is *skewed* ahead of the remote node's state (as of the last message from it).

One crucial thing that Theorem 1 does *not* tell us is what particular state the nodes will actually quiesce *to*. At this stage we cannot derive a meaningful formula for that, because the particular state that will result will depend on the particular function that is used for δ and the particular skew at the time of each received transaction.

We *can* however glean a bit of insight into the matter from ACC-1's code:

1) When a transaction is issued locally, that literal transaction is applied to the local replica.
2) When a transaction is issued remotely and later received locally, it is transformed using the skew sequence and f_A (which uses δ) to account for the skew, and the result is applied to the local replica.

We can therefore conclude that the replica state to which a node quiesces is the one which results from the application of all transactions that were issued in the system—with each node's transactions being in the correct relative order and with the remote node's transactions being interleaved and transformed as above.

This basic system is only one piece of the full, general ACC-1 system, but it is sufficient to present a real-world application.

IV. Application: Real-Time Collaborative Text Editing

Suppose that we want to design a system for allowing two people on opposite sides of the world to collaboratively edit the same text file in real-time over the Internet. That is, we want a program whereby our two users can view and edit the same file on their screens and where edits made by one person become visible on the other person's screen in real-time. Suppose further that we want the two users to be able to make edits simultaneously without needing to coordinate in advance.

Despite the lack of any reference here to a database or transactions, this situation fits our model well. The text file is the "database", and the two copies of it are the replicas. The edits made to the text file are the "transactions", which we will treat as consisting of simply "insert string S at character position N", "delete C characters at character position N", and sequences of transactions of these two types representing an atomic sequence of operations. The $*$ operator therefore simply concatenates such sequences. This is sufficient to implement typing, pressing backspace, pressing delete, pasting from a clipboard, and replacing a selection (via an atomic delete and insertion).

It is clear that this system will need some form of concurrency control. Otherwise, whenever the two users concurrently make a change, they will be applied in a different order at each end and the results will not be guaranteed to be equivalent. For example, if user A issues "insert FOO at 10" and user B concurrently issues "insert BAR at 20", the result on user A's machine will be that "FOO" is at position 10 and "BAR" is at position 20, but on B's machine it will be that "FOO" is at position 10 and "BAR" is at position 23 (because when "FOO" gets inserted, it moves "BAR" forward by three character positions).

The simple solution from traditional concurrency control would be to acquire a distributed lock whenever a transaction is to be issued. That trivially solves the problem of ensuring a coherent final state. However, given that our two users are collaborating with each other from halfway around the world, it will probably introduce a significant noticeable delay in the responsiveness of the program. More precisely, after pressing a key, there will be up to one round-trip worth of network delay before the screen is updated—probably unacceptable. One fix might be to lock only parts of the documents and to do so for longer periods of time so that successive edits to the same area experience no wait, but such heuristic approaches will never completely eliminate the problem.

The alternative is to use ACC-1. Consider choosing $T' = T$ and $\kappa(A) = A$ and implementing δ according to the following functional-like pseudo-code. Note that `I` refers to the identity transaction (i.e., a no-op), which in our chosen T can be simply $\{\}$ (i.e., an empty sequence of operations to perform).

```
delta(E1 as 'insert S1 at N1', 'insert S2 at N2') =
  if N1 > N2 then 'insert S1 at N1+length(S2)'
  else if N1 < N2 then E1
  // Else the insertions are at the same position.
  // Nullify them by inverting the applied one.
  else 'delete length(S2) chars at N2'
delta(E1 as 'insert S1 at N1', 'delete C2 chars at N2') =
  if N1 >= N2+C2 then 'insert S1 at N1-C2'
  else if N1 <= N2 then E1
```

```
  // Else we were going to insert at a position that was deleted.
  // Do nothing.
  else I
delta(E1 as 'delete C1 chars at N1', 'insert S2 at N2') =
  if N1 >= N2 then 'delete C1 chars at N1+length(S2)'
  else if N2 >= N1+C1 then E1
  // Else delete what we had before, plus the inserted thing.
  else 'delete C1+length(E2) chars at N1'
delta(E1 as 'delete C1 chars at N1', 'delete C2 chars at N2') =
  if N1 >= N2+C2 then 'delete C1 chars at N1-C2'
  else if N2 >= N1+C2 then E1
  // Else delete whatever has not already been deleted.
  else
    if N2 <= N1 then
      if N2+C2 < N1+C1 then 'delete C1-(N2+C2-N1) chars at N2'
      else I
    else
      if N1+C1 < N2+C2 then 'delete N2-N1 chars at N1'
      else 'delete N1-N2 chars at N1'
delta(E1, E2*E3) = delta(delta(E1, E2), E3)
delta(E1*E2, E3) = delta(E1, E3)*delta(E2, delta(E3, E1))
```

Note that because `delta(E1, E2*E3)` and `delta(E2*E3, E1)` will make a common recursive call (`delta(E1, E2)`), it is desirable in practice for the two calls to `delta` in the ACC-1 pseudo-code to actually be computed simultaneously in one call for better efficiency.

A case-by-case analysis of the above pseudo-code will show that it satisfies the symmetry criterion.[13] The first two cases of each outer `if`-ladder correspond to the situation where the edits are to different areas of the document and thus do not conflict. The third case of each is the case where the edits conflict.

Intuitively, this definition of $\delta(A, B)$ will transform A to assume that B was applied without the knowledge of the person who sent A. The result it produces is what it thinks the sender would have wanted to send instead of A if they had known about B. The end result will typically be what the users expect. For example, if two users concurrently edit separate sections of the document, the above definition of δ will produce the same result that one would get if the editing was done off-line and the two versions were later merged.[14]

Using the above definition of δ, ACC-1 can be used as the concurrency control system for a real-time collaborative text editor. Since processes that issue transactions to ACC-1 do not incur any delay due to network communication, the text editor's GUI will have the same responsiveness as a normal text editor, regardless of the network latency.

It is worth noting that this is not a made-up problem. ACC-1 was used for exactly this purpose in a program called XCDE (eXtreme Collaborative Development Environment) by Larry Chen, Andrew Craik, Tom Levesque, and Tristan Schmelcher. It added real-time collaborative text editing to the open-source Eclipse IDE via a plug-in, and was the original motivation for ACC-1.[15] Work is now underway to include XCDE's technology in the Eclipse Communication Framework. XCDE does not use precisely what has been presented above, though, because it also incorporates other ACC-1 techniques that we will introduce later.

Before moving on, two things about this application are worth noting.

[13] We elide such an analysis because there are many cases and they are all fairly boring.

[14] This "expectedness" property follows from the meaning of the skew sequence and the behaviour of the above δ.

[15] Since then the XCDE authors have discovered that asynchronous concurrency control has been studied before in this particular context under the name "operational transformation". [2]

A. Elimination of the Undoability and Group Assumptions

The reader might observe that this system does not satisfy the property that T be a group under $*$, because the transactions do not have inverses.[16] However, that does not matter. Recall that we earlier made the following statement after assuming that transactions are invertible:

> ... it should be possible to make most systems satisfy this by having the database back-up any state that is unrecoverably overwritten. A special transaction for resurrecting the backed-up state could then be used to undo transactions that otherwise would not be invertible.

Suppose that we were to do as we proposed; we would thus introduce a new kind of transaction: "undo t", where "t" is some other transaction description. It would be defined as follows: when applied to a text file, if that text file's state is equivalent to the application of a sequence of transactions that ends in "t", then "undo t" undoes the effects of t. Otherwise—i.e., if the state is not equivalent to the application of any sequence of transactions that ends in "t"—"undo t" records that if the sequence of transactions that is applied in the *future* ever becomes equivalent to one that *starts* with "t", then the effects of "t" should be undone and the record removed. "undo t" will thus be a true inverse of "t", so T will form a group under $*$ and all the theorems will apply.

Now observe the following: even though the equations and proofs in this paper involve the inversion operator, the ACC-1 pseudo-code itself does not. Further, the above definition of δ does not either. Therefore, in practice our real-time collaborative text editor will never instantiate the "undo t" transaction description. Therefore it will also never apply it to the text file, and therefore the extra information that it maintains to enable that will never be used. Thus, this "fix" to our system does not actually change its behaviour, and therefore does not actually need to be made.

This illustrates an important point: in practice, T does not need to form a group under $*$ at all. It does not need to contain inverses for any of its transactions (nor even an identity transaction, because the ACC-1 pseudo-code does not use that either) unless the definition of δ (or κ) needs to construct them. This is true because it is always possible to add imaginary "undo" transactions as above (and a trivial identity transaction), and doing so would never change the behaviour if those objects had not already been needed.

B. Level of Consistency

The reader might note a suspicious behaviour of this system: when two insertions are concurrently attempted at the same character position, they will be nullified so that neither is present in the final result. This is necessary here because the δ implementation uses character positions and edit types to decide which of two concurrent edits should adjust to accomodate the other. Thus, when the character positions and edit types are both the same, there is a tie and the only solution is to nullify the edits.[17]

This is perhaps acceptable for a real-time collaborative text editor,[18] but that it appears tricky to avoid would seem to conflict with our claim in the abstract that "ACC-1 can provide full consistency for any distributed system". The paragon of consistency is usually sequential consistency, and erasing two concurrent transactions from history is certainly not compatible with it.[19] We are, therefore, certainly not providing "full" consistency in this example, and the reader might wonder how it can be that it is always possible to do so. That is the subject of the next section.

[16] A deletion transaction cannot be undone by any single other transaction. Further, although every insertion transaction can be undone by a single deletion transaction, the insertion does not always undo that deletion, so they are not true inverses.

[17] Actually, this is not entirely true; δ could lexicographically order the insertion strings and have the edit with the smaller one adjust to accomodate the edit with the larger one. When the insertion strings are equal too, the edits are equivalent and thus commute.

[18] Though we will later "fix" it anyways.

[19] As it happens, this particular system is incompatible in other ways too. In fact, it does not *desire* true sequential consistency, because the transactions that users' computers issue contain the location of the user's cursor at that moment in time as a hard-coded value. Thus whenever that value becomes out-of-date due to skew, it is desirable to apply a different transaction than what the computer actually specified.

V. Sequential Consistency

We stated in the abstract that ACC-1 can always provide full consistency to any distributed system. The paragon of consistency is usually sequential consistency,[20] which is defined as follows.

> [A system is defined to be sequentially consistent if and only if] the results of any execution are the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program. [3]

In the context of our distributed system, the "processors" of course correspond to the two nodes in the network. The "operations" of the processors are the transactions that the nodes execute on their replicas of their own accord, and the "result" is the database state once all operations have been issued and all skew sequences have become empty (which from before will be the same at every node). The "order specified by its program" corresponds to the order of issue.

It is easy to see that sequential consistency will never *automatically* be satisfied, because it is always possible to choose $T' = T$, $\kappa(A) = A$, and $\delta(A, B) = B^{-1}$. This will trivially satisfy the symmetry criterion. (Both sides are I.) However, an ACC-1 system that uses it will not satisfy sequential consistency. For example, if two transactions are concurrently issued at the two nodes, the final result will be that the state has not changed from when it started. (Essentially, these choices have the semantics that transactions that are concurrent with each other are nullified.) Sequential consistency would require that the final result be that which is obtained from executing the two transactions in some order.

Clearly then, to get sequential consistency, we must be more restrictive in how ACC-1 can be used. As it happens, we are going to need to use a slightly modified version of the fundamental 2-node system, which we now present.

A. Extension to the 2-Node System with Primacy

Let $(T, *)$ be any transaction group (and I its identity element), and assume that we have defined some T' and some $\kappa : T' \to T$, but that instead of δ we have some function $\Delta : (T' \times T' \times \mathbb{B}) \to T'$ satisfying the property that

$$\kappa(B') * \kappa(\Delta(A', B', b)) = \kappa(A') * \kappa(\Delta(B', A', \neg b))$$

We will call this the semisymmetry criterion. The function Δ can be thought of in the same way as the normal δ. The difference is that it requires some asymmetry to be injected via an added boolean value in order to satisfy its criterion. Thus the above amounts to assuming that we have some situation where we want to use ACC-1 but where the objects don't quite satisfy the fundamental 2-node system's requirements.[21]

Let $T_\sigma = T \times \mathbb{B}$ and define $*_\sigma : T_\sigma \times T_\sigma \to T_\sigma$ for all $A \in T$, $B \in T$, $a \in \mathbb{B}$, and $b \in \mathbb{B}$ as

$$(A, a) *_\sigma (B, b) = (A * B, a \oplus b)$$

where $\oplus$ is the boolean exclusive-or operator.

Observe that $(T_\sigma, *_\sigma)$ is a group. Its identity element is $(I, false)$ and the inverses are given by $(A, a)^{-1} = (A^{-1}, a)$.

Intuitively, this new transaction group corresponds to adding a flag to the replica states and transactions of the original system, where setting the flag on a transaction means that the flag value on the replica to which it is applied should be switched. We will later see that the flag is set on at most one of the two replicas at a time.

[20] Though we will see later that, for some applications of ACC-1, it can be desirable to provide something even stronger.

[21] We allow for said situation to have defined an arbitrary κ, but in fact we will not make use of that ability in this paper.

Note that if we let $\sigma = (I, true)$ and define $f_\sigma : T \rightarrow T_\sigma$ by $f_\sigma(A) = (A, false)$, then $f_\sigma(T) \cup \{\sigma\}$ generates T_σ (hence the σ subscript notation). If we assume that in practice the sequencing operator just constructs lists, then we can implement this new transaction group by simply (1) creating a new transaction σ that switches the flag value on the replica to which it is applied and (2) extending the sequencing operator $*$ to accept it. (The addition of a false flag to the existing transactions is in practice unnecessary, since it adds no information.) Further, define a σ-transaction as one of the form $(A, true)$. Then the σ-transactions correspond to those sequences in which there are an odd number of occurrences of σ (in practice there will be at most one).

Now let $T'_\sigma = T' \times \mathbb{B} \times \mathbb{B}$ and define $\kappa_\sigma : T'_\sigma \rightarrow T_\sigma$ as $\kappa_\sigma(A', a_1, a_2) = (\kappa(A'), a_1)$ and δ_σ as follows.

$$\delta_\sigma((A', a_1, a_2), (B', b_1, b_2)) = (\Delta(A', B', a_2), a_1, a_2 + b_1), b_2 = \neg a_2, (\neg a_1) + a_2, (\neg b_1) + b_2$$

Intuitively, the second flag that we have added will represent the value of the replica's flag at the node that issued the transaction, and the above definition of δ_σ uses that to generate the asymmetry that Δ requires. For clarity, call the first flag the "σ-flag" and the second one the "extended flag".

Note that due to the conditions on the above definition, δ_σ is only defined over a *subset* of the $T'_\sigma \times T'_\sigma$ domain that the fundamental 2-node ACC-1 system expects. We will see though that this does not matter, because none of the undefined cases will occur in practice.

First we confirm that δ_σ satisfies the symmetry criterion in those cases when it is defined.

The $\delta_\sigma((A', a_1, a_2), (B', b_1, b_2))$ term is defined whenever $b_2 = \neg a_2$, $(\neg a_1) + a_2$, and $(\neg b_1) + b_2$ are true.

The $\delta_\sigma((B', b_1, b_2), (A', a_1, a_2))$ term is defined whenever $a_2 = \neg b_2$, $(\neg b_1) + b_2$, and $(\neg a_1) + a_2$ are true, which together are equivalent.

Thus the equation's truth value is defined whenever $b_2 = \neg a_2$, $(\neg a_1) + a_2$, and $(\neg b_1) + b_2$ are true. So,

$$\begin{aligned}
&\kappa_\sigma(B', b_1, b_2) *_\sigma \kappa_\sigma(\delta_\sigma((A', a_1, a_2), (B', b_1, b_2))) \\
&\qquad = \kappa_\sigma(A', a_1, a_2) *_\sigma \kappa_\sigma(\delta_\sigma((B', b_1, b_2), (A', a_1, a_2))) \\
&\Leftrightarrow (\kappa(B'), b_1) *_\sigma \kappa_\sigma(\delta_\sigma((A', a_1, a_2), (B', b_1, b_2))) \\
&\qquad = (\kappa(A'), a_1) *_\sigma \kappa_\sigma(\delta_\sigma((B', b_1, b_2), (A', a_1, a_2))) \\
&\Leftrightarrow (\kappa(B'), b_1) *_\sigma \kappa_\sigma(\Delta(A', B', a_2), a_1, a_2 + b_1) \\
&\qquad = (\kappa(A'), a_1) *_\sigma \kappa_\sigma(\Delta(B', A', b_2), b_1, b_2 + a_1) \\
&\Leftrightarrow (\kappa(B'), b_1) *_\sigma (\kappa(\Delta(A', B', a_2)), a_1) \\
&\qquad = (\kappa(A'), a_1) *_\sigma (\kappa(\Delta(B', A', b_2)), b_1) \\
&\Leftrightarrow (\kappa(B'), b_1) *_\sigma (\kappa(\Delta(A', B', a_2)), a_1) \\
&\qquad = (\kappa(A'), a_1) *_\sigma (\kappa(\Delta(B', A', \neg a_2)), b_1) \\
&\Leftrightarrow (\kappa(B') * \kappa(\Delta(A', B', a_2)), b_1 \oplus a_1) \\
&\qquad = (\kappa(A') * \kappa(\Delta(B', A', \neg a_2)), a_1 \oplus b_1) \\
&\Leftrightarrow (\kappa(B') * \kappa(\Delta(A', B', a_2)), b_1 \oplus a_1) \\
&\qquad = (\kappa(A') * \kappa(\Delta(B', A', \neg a_2)), b_1 \oplus a_1) \\
&\Leftrightarrow \kappa(B') * \kappa(\Delta(A', B', a_2)) = \kappa(A') * \kappa(\Delta(B', A', \neg a_2)) \\
&\Leftrightarrow true \qquad\qquad \textit{By semisymmetry criterion}
\end{aligned}$$

Thus the equation holds. Therefore, provided that δ_σ is only ever computed by ACC-1 for the case in which it is defined, T_σ, T'_σ, κ_σ, and δ_σ form a valid instantiation of the fundamental 2-node system.

To ensure that this holds, we constrain the use of this instantiation of the system as follows:

- The flags on the two nodes' replicas must be initialized to opposite values. i.e., one replica must be initialized to $(A, true)$ (for some $A \in T$), while the other must be initialized to $(B, false)$ (for

some $B \in T$, where typically $A = B$). Recall that the fundamental 2-node system did not require the initial states to be equal, so this is valid. (The theorems were for the *differential* state; i.e., the transaction that would bring the state from its initial state to its final state.)

- Each node makes sure that it only issues a transaction with the flag set (i.e., $(A, true)$ for some $A \in T$) if its replica state at that moment also has its flag set. i.e., nodes cannot turn on their own replica's flag; they can only turn it off.
- Whenever a node issues any transaction $(A, a_1) \in T_\sigma$, the extended transaction $(A', a_1, a_2) \in T'_\sigma$ ($\kappa(A') = A$) that it sends to the local ACC-1 component has a_2 equal to the current value of the flag on that node's replica.

It can be seen that this is sufficient to prevent the undefined cases of δ_σ from occuring. Observe:

At any time, at most one replica will have its flag set. Further, when neither have their flag set, there will be exactly one transaction in transit that has its σ-flag set, and at all other times there will not be any such transaction in transit. This follows from the facts that:

1) The replicas start with opposite flag values.
2) Only a replica with its flag set can send a transaction with the σ-flag set.
3) Such a transaction will turn off the sender's flag and turn on the receiver's flag.
4) The transformations of such a transaction at the receiver will preserve its flag value.

By the same token there will be at most one such transaction in a given node's skew sequence, and it will never receive such a transaction so long as there is such a one in its skew sequence.

We can now see that, since the extended flag in an extended transaction is always set to the sender's current flag value and is only changed when synchronizing past a transaction with the σ-flag set, the $(\neg a_1) + a_2$ and $(\neg b_1) + b_2$ conditions will always hold when computing δ_σ, because the conditions can be read as saying "the σ-flag cannot be set if the extended flag isn't".

Further:

1) If there is an extended transaction in a node's skew sequence that has the σ-flag set, then it and all entries in the skew sequence that precede it will have their extended flag equal to the opposite of the local replica's current flag.
2) All other entries in a skew sequence will have their extended flag value equal to the replica's current flag value.
3) Any transactions that are received while the skew sequence contains one with the σ-flag set will have extended flag value equal to the local replica's current value, and thus the opposite value of the extended flag of those transactions at or before the one in the skew sequence with the σ-flag set.
4) Thus for the δ_σ calls for those transactions, the $b_2 = \neg a_2$ condition will be satisfied.
5) Further, after synchronizing past the transaction with the σ-flag set, the extended flag on the received transaction will be the opposite of the local replica's current value, and thus the opposite value of the extended flag of all of the remaining transactions in the skew sequence.
6) Thus for all those transactions, the $b_2 = \neg a_2$ condition will also be satisfied.
7) In the case where there is no transaction in the skew sequence with the σ-flag set, any received transactions will have the opposite extended flag value and $b_2 = \neg a_2$ will be satisfied.
8) Lastly, whenever a transaction with the σ-flag set is received, everything in the skew sequence will have its extended flag value switched so that these assumptions continue to hold.[22]

Thus all the constraints hold all the time.

Therefore, since all of the algorithm's calls to δ_σ will be defined, Theorem 1 will apply.

From that theorem, we have that the system always quiesces to a state where the differential states are equal. Thus, from the definition of $*_\sigma$, we can see that if the initial states (excluding the flag values) are

[22]Note that extended flag values are so tightly constrained by these properties that in practice it is not actually necessary to transmit them over the network or store them in the skew sequence. They can always be predicted from the local replica's flag value and the position of any transaction in the skew sequence with the σ-flag set.

equal, then so too are the final states.

We thus now have a version of ACC-1 that we can opt to use over the original one whenever we wish for synchronization semantics to be asymmetric; i.e., for one node to synchronize its received transactions using different behaviour than is used at the opposite node. We need only define T, T', κ, and Δ (instead of δ) and can then apply the system that we have presented in this section. We will see that this allows us to achieve sequential consistency.

Further, by having the node with its flag set (call it the "primary node") issue the σ transaction, we can have the nodes swap their synchronization behaviours on-the-fly. (We will not use this until much later.)

We call this the "2-node system with primacy", because the flag on a replica denotes a position of superiority or "primacy" over the other node, since only the node with its flag set can cause synchronization behaviours to be swapped. We will also typically define Δ in ways that favour the node that has primacy.

Note also that this system is strictly more general, because one can always choose to implement Δ by ignoring the added boolean flag. This results in the fundamental 2-node system with no significant overhead.

B. Aside: Breaking Ties

Before moving on to show how to achieve sequential consistency, as a brief aside we will first show another minor application for primacy. Recall that in our definition of δ for a real-time collaborative text editor, concurrent edits to the same location had to be nullified because there was no way to choose which one to apply first. With primacy, this short-coming can be eliminated; we can choose to have the non-primary node's edit appear to happen first and the primary node's edit appear to happen second (so that it ends up at the specified position and the other string is moved forward). This would correspond to the following modified pseudo-code:

```
Delta(E1 as 'insert S1 at N1',
      'insert S2 at N2',
      P) =
  if N1 > N2 or (N1 = N2 and !P) then
    'insert S1 at N1+length(S2)'
  else E1
Delta(E1 as 'insert S1 at N1',
      'delete C2 chars at N2',
      P) =
  if N1 >= N2+C2 then 'insert S1 at N1-C2'
  else if N1 <= N2 then E1
  // Else we were going to insert at a position that was deleted.
  // Do nothing.
  else I
Delta(E1 as 'delete C1 chars at N1',
      'insert S2 at N2',
      P) =
  if N1 >= N2 then 'delete C1 chars at N1+length(S2)'
  else if N2 >= N1+C1 then E1
  // Else delete what we had before, plus the inserted thing.
  else 'delete C1+length(E2) chars at N1'
Delta(E1 as 'delete C1 chars at N1',
      'delete C2 chars at N2',
      P) =
  if N1 >= N2+C2 then 'delete C1 chars at N1-C2'
  else if N2 >= N1+C2 then E1
```

```
  // Else delete whatever has not already been deleted.
  else
    if N2 <= N1 then
      if N2+C2 < N1+C1 then 'delete C1-(N2+C2-N1) chars at N2'
      else I
    else
      if N1+C1 < N2+C2 then 'delete N2-N1 chars at N1'
      else 'delete N1-N2 chars at N1'
Delta(E1, E2*E3, P) =
  Delta(Delta(E1, E2, P), E3, P)
Delta(E1*E2, E3, P) =
  Delta(E1, E3, P)*Delta(E2, Delta(E3, E1, !P), P)
```

This is the actual synchronization method that XCDE uses (neglecting its many extensions, such as directory trees containing multiple text files, dynamic file/directory creation/deletion, user information and cursor position highlighting, and annotations). However, since the "priviledge" of inserting second is not particularly valuable, XCDE does not bother implementing the ability to exchange primacy. We will later see systems where it is required.

This is still not the complete ACC-1 system used by XCDE; XCDE is a multi-node tree topology, the theory for which we have not yet introduced.

C. Extension to Sequential Consistency

It is now relatively easy to achieve sequential consistency for any distributed system. Given any transaction group T, make the following choices for the 2-node system with primacy.

$$\begin{aligned} T' &= T \\ \kappa(A) &= A \\ \Delta(A, B, b) &= B^{-1} * A * B \ if \ b, \ else \ A \end{aligned}$$

The above definition for Δ does satisfy the semisymmetry condition. (If $b = true$ then both sides are $A * B$. If $b = false$ then both sides are $B * A$.)

It can easily be proven that these choices satisfy the property that, at all times, a replica's state is that which results from some interleaving of precisely all transactions that have so far been issued locally or received.

To see this, it is enough to simply observe that the primary node always applies its literal received transaction without modification. Thus the result there is indeed some interleaving of the transactions issued or received thus far (specifically, the same interleaving as that in whose order they were issued or received).

Therefore the final result there is sequentially consistent, and Theorem 1 already guarantees that the state at the other end is the same.

The situation is slightly less straightforward when primacy exchange is considered, but sequential consistency holds nonetheless. To see this, observe that when a σ-transaction going from $\neg N$ to N reaches N, the committed state there will already be equal to the state of $\neg N$ before it sent the σ-transaction (because of Theorem 1), and the skew sequence will simply contain the transactions that were issued at N and that were not yet part of $\neg N$'s state when it sent the σ-transaction. Thus, after N's primacy flag and the extended flags in its skew sequence are flipped, the situation will be no different than if the system had (1) quiesced before N issued the transactions in its skew sequence, (2) exchanged primacy off-line (which clearly is sound), and then (3) continued by immediately issuing those transactions. Thus the final result

at N will be a sequentially consistent result for the transactions issued after the exchange, all applied to a sequentially consistent result for the ones issued before it, and so it is sequentially consistent overall.

Note that when using this exact instantiation for the 2-node system with primacy, several optimizations can be made beyond those mentioned so far. First, there is no need to maintain any entries in a skew sequence that have the extended flag set and the σ-flag not set, because they will never be used to synchronize a received transaction (since $\Delta(A, B, false)$ ignores B). It is thus possible for the primary node to forego appending anything to its skew sequence at all (until it relinquishes primacy), provided that the non-primary node does not send acknowledgements (until it receives primacy). In fact, the resulting system is so simple that it is a bit of a stretch to call it an ACC-1 instantiation at all.

Note further that the ability to exchange primacy is not necessary to achieve sequential consistency; one node can simply remain the primary node forever (in which case the system is *definitely* trivial). However, we will later see a situation where both sequential consistency and primacy exchange are necessary.

1) Implications for Invertibility: The reader might notice that, since the above definition of Δ involves the inverse operator, it would seem to be necessary for every system that desires sequential consistency to actually implement the ability to invert all transactions. This would indeed be true if $B^{-1} * A * B$ were always executed naively by undoing B, executing A, and re-executing B, but there is no need for that to be so. Rather, Δ can return any group-theoretically equal transaction, and it further can be executed in any functionally-equivalent manner.

For example, suppose that A is "x <− x + 3", B is "y <− x", and $b = true$. The definition of Δ seems to say that the system must restore the previous value of "y", add 3 to "x", and then write "x" to "y". However, this is equivalent to simply only doing the latter two operations, because the restored value for "y" would not be used during those two operations and would be overwritten by the end of them. Thus, in this situation, $B^{-1} * A * B$ can be executed by simply doing $A * B$,[23] and therefore the system will not actually need to resurrect the previous value for "y".

Some systems that demand sequential consistency may be able to use such optimizations to completely eliminate the need to implement invertibility, but that will probably be rare. Most non-trivial systems will probably need to support true inverses in at least some cases. For example, if A above was instead "x <− x + 3; z <− y", then there is no avoiding resurrecting the old value of "y". However, such situations may prove to be the minority. For example, if A and B commute, then $B^{-1} * A * B = A$, so Δ can just return A. This will occur whenever the transactions operate on disjoint data sets, which we would expect to be the norm in a well-parallelized system.

Further, even if a system may potentially need to invert every transaction that it applies to its replicas, there will still be benefits in performing optimizations such as those above. This is because otherwise the $b = true$ case of Δ will always increase the amount of work that must be done to execute the transaction that it synchronizes; executing the final transaction on the replica will involve an additional amount of work equal to that that is required to undo and redo everything in the skew sequence. In systems that issue transactions very frequently (relative to the amount of network delay), the skew sequence will on average be very long and thus this added work may decrease the computational efficiency of the system to the point where it eclipses the network delay of a traditional concurrency controller as the limiting factor. Thus optimizations like those above should be made whenever convenient. The expected prevalence of commuting transaction pairs should then prevent pathological usage patterns from arising in practice.

VI. Multinode Systems

As presented thus far, ACC-1 can only be applied to 2-node distributed systems. Clearly this is a severe shortcoming. However, in this section we show how to build n-node ACC-1 systems out of what we have already seen.

[23] Though they are not group-theoretically equal.

A. Tree Topologies

For this section we consider n-node systems where the graph of network connections forms a tree. That is, in a graph of the network where the nodes are the computers and the edges are the bi-directional connections between them, we assume that there is a unique path between any two nodes. Note that we do *not* assume that there is any agreed-upon parentage relation. All assumptions from the previous systems continue to apply.

1) Definition: The system works by running an instance of a 2-node ACC-1 system on every connection. Either the fundamental or primacy system can be used, but we will assume the latter since it is strictly more general.

Let $e(N)$ denote the number of edges that node N has in the graph of network connections. Then each node N is involved with $e(N)$ instances of the fundamental system.

- The $e(N)$ ACC-1 instances at each node share that portion of their replica state that comes from T (i.e., they share the A in (A, a)), but they all have separate flags and separate skew sequences. Thus having primacy is a per-link property.
- As before, the per-link flags at each end of a connection are initialized to opposite values.
- The handling of each of the three event types for each of the $e(N)$ instances is exactly as given for the 2-node system.
- User processes at a node are given the ability to either (1) issue some $A' \in T'$ (which gets mapped to $(A', false)$) or (2) send σ on a specified link that has its primacy flag set.
- When a user process at a node wants to issue some transaction $A' \in T'$,
  1) the node executes $\kappa(A')$ on the shared replica, and
  2) for each ACC-1 instance, the node gives $(A', false, a)$ to that instance, where a is its primacy flag's current value.
- When a user process at a node wants to relinquish primacy,
  1) that process must specify which link is to relinquish primacy (and that link must have its flag set, as before), and
  2) the node then flips that link's flag and gives $(I, true, a)$ to that link's ACC-1 instance, where a is its primacy flag's current value.
- When an ACC-1 instance gives some transaction (A', b, a) to the local node,
  1) the node flips that instance's flag if $b = true$,
  2) executes $\kappa(A')$ on the replica, and
  3) for each of the $e(N) - 1$ other ACC-1 instances, the node sends $(A', false, a)$ to that instance, where a is its primacy flag's current value.
- Note that all three of the above procedures must be completely atomic, because each instance of the 2-node system requires that it be atomic together with the event that causes it or that it causes.

2) Analysis: Due to the way that the system has been constructed, all the analysis of the 2-node system continues to apply for each instance of it in a tree system. This is because (1) the semantics of the message passing between the local node and each ACC-1 instance are unchanged from that ACC-1 instance's perspective, and (2) the use of the primacy extensions still satisfies the constraints that it adds.

As a result, the theorems from before are directly applicable to each instance. However, most of the notation is no longer adequate, because identifying a node no longer uniquely indentifies an ACC-1 instance. To fix this, each definition that used to accept a node N an an argument will now accept two nodes N and M (in that order) that must be connected. The ACC-1 instance that it refers to is N's instance for its link with M.

We can thus rewrite Theorem 1 as follows.

Theorem 2: For all N, all M to which N it is connected, and all e,

$$C(N, M, e) = B(M, N, k(N, M, e)), \tag{5}$$

$$F(N, M, e) \; contains \; exactly \; L(N, M, e) \; transactions \\ and \; L(M, N, k(N, M, e)) \; acknowledgements, \tag{6}$$

and $\forall i \in [1, L(N, M, e)]$,

$$S(N, M, e)[i] = f_A(F(N, M, e)[p(N, M, e, i)], X) \\ where \; X = g(F(N, M, e)_{1..(p(N,M,e,i)-1)}, S(M, N, k(N, M, e))) \tag{7}$$

As described above, there is no need to re-prove the theorem because the previous proof still applies. However, it remains to determine from this theorem how the tree system as a whole will behave.

Consider some node N. Suppose that, at some moment in time, all of its skew sequences are empty.

Then by Theorem 2 equation (5), its state is a past state of every node M to which it is connected (specifically, the state at each M as of the last message from it).

If we suppose further that each M's skew sequence for all of its other connected nodes was empty at the time of that past state, then Theorem 2 equation (5) further tells us that N's state is a past state of all of those nodes too (specifically, the state as of the last message from each of them that had arrived at M as of the last message from it).

Generalizing this logic, we can make the following inductive statement:

> For any two nodes N_1 and N_2 (not necessarily connected), the state at N_1 is a past state of N_2 (specifically, its state as of the last message from it that has been communicated to N_1) whenever:
>
> 1) N_1's skew sequence in its instance for the connected node M that lies along the path to N_2 is empty, and
> 2) M's state as of the last message from it was a past state of N_2 (as determined by this definition).

Thus, when every skew sequence in the system is empty, all the replica states are equal. A tree-structured ACC-1 system thus shares the 2-node system's property that the nodes never get out-of-sync.

Also like before, when it comes to the question of what particular state the system will quiesce to, we cannot say anything of much meaning on the matter without knowing how Δ is defined by the particular application. Like before, locally issued user transactions will be applied directly to the replica, while remotely issued ones will be potentially transformed using Δ. However, the latter situation is more complex than before, because the transformation will have happened once for every link that the transaction traversed.

3) Sequential Consistency: In a multi-node tree system, simply using the instantiation given in §V-C is not sufficient to achieve sequential consistency. A further restriction is necessary: at all times, each node can have at most one primacy flag value set to false.

To see that this is sufficient, observe that, since we know that the network graph is a tree and that the two ends of a link can only both have false primacy flags if a σ-transaction is in transit between them, it follows from the above added restriction that, if no such transaction is in transit, then there exists exactly one node with all its primacy flags set to true. (There must exist one because otherwise following links with false primacy flags results in a cycle, and there cannot exist more than one because then somewhere along the path between any two of them would be a node with more than one false primacy flag.)

Call that node the "global primary node" (and call the property "global primacy"). The transactions that are executed at it will be the literal transactions that are issued in the system, because they will traverse all links with a false extended flag and thus not be changed in the process. Therefore the final result at the global primary node will be sequentially consistent. As before, the previous general analysis tells us that the final state everywhere else is the same.

The situation is not much more complex when considering primacy exchange, because there will be at most one σ-transaction in transit. To see this, observe that if there is one in transit, then both ends of the

link it is traversing must have false primacy flags for that link. Thus any path from one of them to any other node on its side of that link will only cross links from a side that has a true primacy flag, because otherwise there is a node along the path that has more than one false primacy flag. Thus there are no other σ-transactions in transit, because every link can be traversed by such a path and a σ-transaction can only be traversing a link where both ends have false primacy flags. Thus the transactions that arrive from the rest of the system are the literal issued ones and the primacy exchange is no different than in the 2-node case.

The above restriction is sufficient to get sequential consistency. It is also easy to make it be obeyed in practice; simply initialize the system in a way that satisfies it[24] and thereafter only allow a user process to relinquish primacy on a link at some node if each link there currently has primacy (i.e., if the node has global primacy).

B. Complete Topologies

Tree topologies have a potential shortcoming in that every link is the *only* link that connects its two sides of the network. This may be a drawback in some situations, such as those where the links are physical and where we could thus increase network capacity if only we could add more links.[25] We therefore consider "complete" topologies—i.e., those where the network graph is a complete graph. A complete graph is one in which every pair of nodes is connected, so we are thus considering the case where there is a dedicated link for all communication between each pair of nodes.

Assume that we have n computers that we want to connect into a complete topology. However, suppose that, for lack of any ACC-1 theory for complete topologies, we decide to resort to a tree topology. Further, the particular tree topology that we choose is one in which each of our n computers is connected to a new computer that will simply serve to route transactions between its links (i.e., it will not issue any transactions of its own, nor serve any other purpose). We will call it the server, and the other nodes the clients.

Now imagine that the computer that we use for the server has the magical property that all messages that it sends arrive instantaneously at their destination clients. Clearly this is impossible in the real world, but neither ACC-1 itself nor the proofs of its properties rely on any specific values for the network delay in either direction. Thus ACC-1 works for any amount of network delay, so in particular it will work in the case of zero network delay in one direction. As a result, ACC-1 could be used in this imaginary system and all of the previous results would apply.

We now show that it is possible to virtualize this magical server. That is, it is possible to remove the actual server node and have each client independently simulate its operation so as to produce the same results. To see this, suppose that we change the clients so that, instead of sending their messages to the server, they send them to every other client along a dedicated link (which forms a complete graph). Every client will then know all of the messages that the server would have seen (because each automatically knows all of the ones that it sent, and the above ensures that it knows all the others too). Provided that the clients can agree on an order in which the virtual server will "receive" those messages, they can all independently simulate the server's actions and get identical results.

To see this, suppose that we have some way to decide how to order the receive events at the virtual server. Then each client can operate as follows:

1) On boot, initialize the data that a physical server node would use.[26] This is the virtual server's state.
2) Whenever it is determined that a certain event is the next event that shall happen at the virtual server, simulate its effect on the virtual server's state.

[24]This is easy if there is a defined parentage relation such as a server-client hierarchy.

[25]Extra links are also desirable for fault-tolerance, but we have not gotten there yet.

[26]This is in addition to the normal client data, which must also be initialized.

3) Further, if the simulation of the virtual server indicates that it would have sent a message to us, then immediately simulate the reception of that message (because, if using the imaginary magical server, it would have arrived instantaneously).

From the reasoning presented above, this would result in a system whose behaviour would be externally indistinguishable from a tree system. Thus all the same reasoning would hold and ACC-1 would work.

As for the missing piece about deciding on an ordering for the virual server's receive events, totally ordered broadcast algorithms are directly applicable and easily solve the problem. [4] Thus complete topologies are possible.

We note two things before moving on. First, observe that the message streams that a given client sends to all the others are identical (because they are each a copy of what that client would have sent to a server). Therefore this system can be easily adapted into one for a shared-medium network, too.

Second, there is some room for optimization in this design, because in fact it is only the ordering of *transaction* receive events at the virtual server that will affect the final result. This is because acknowledgement receive events only affect the data for their link, and their effect (i.e., removing from the front of the skew sequence) is commutative with the effect of sending a transaction on that link (i.e., adding to the end of the skew sequence). Thus the clients need not agree on when acknowledgement events at the virtual server occur; instead they can each simulate them as soon as they hear about them.[27] Further, many totally ordered broadcast algorithms involve acknowledgement messages of their own, so those could be combined with the ACC-1 acknowledgement messages to further improve efficiency.

C. Mixed Topologies

We briefly note that, since ACC-1 on a complete topology is equivalent to ACC-1 on a certain tree topology, we can embed complete topologies inside tree topologies with no need for any additional design or analysis.

For example, if some or all of our n nodes in §VI-B had had additional links to other parts of a tree network, it would not have affected the analysis. Therefore, we can freely add additional links to the nodes in any complete topology. Further, they can be either normal tree-style links or complete-style links to other complete topologies, because the latter are equivalent.

Essentially this means that there are two kinds of ACC-1 links—2-node links and n-node complete links—and they are interchangeable from the perspective of the routing logic that was presented in §VI-A.1.

Sadly, this approach will not let us run ACC-1 on arbitrary network topologies, because a ring of more than three nodes is neither a tree nor a complete graph. However, in some sense the complete topology on its own will let ACC-1 run on any topology, because each node's messages can simply be routed throughout the network to every other node. Each node can then simulate a virtual server for all the nodes.

VII. Full Consistency

We have seen how to achieve sequential consistency via ACC-1 for any distributed system, but is this really a sufficient level of consistency for all systems? Recall the definition of sequential consistency and observe the following: the only statements that it makes are about the properties of the system's result. It says nothing about intermediate states. Thus, if the external world were to observe the state of the system before the end, sequential consistency does not say anything about what it will see.

What about ACC-1 itself? Is its behaviour actually sensible regardless? No. Suppose that we have two nodes and we issue A at one and B at the other, and then user processes at each node immediately observe the state of the replica. At the first node, the user will see that the replica's state is A, while at

[27] Subject of course to the restriction that any previous transactions from their senders must have already been simulated, or else the simulation corresponds to a tree system with out-of-order message delivery.

the other the user will see that it is B. Later, once the system has quiesced, the final state will be either $A * B$ or $B * A$ (depending on who had primacy). These correspond to the sequential schedulings of $\{A, B\}$ and $\{B, A\}$, respectively. However, in the first sequential schedule it would have been impossible for the second node to ever see the state as B, while in the second sequential schedule it would have been impossible for the first node to ever see the state as A.

We thus have a problem: even when ACC-1 is providing sequential consistency, it is not necessarily providing "full" consistency if the external world is allowed to observe intermediate states.

In this section we will show how to provide full consistency all the time.

A. Definition

It is of course necessary to first decide precisely what "full" consistency actually is. We define it as follows:

Definition 13: A system has *full consistency* (or is *fully consistent*) if and only if, whenever its state is observed by the external world at some node N, that state is equal to (1) the state that was most recently observed at any node plus (2) a sequentially consistent result for (i) all transactions so far issued at N whose effects were not visible in the previous observed state and (ii) all, some, or none of those such transactions issued at other nodes.

Clearly any fully consistent system has the property that recording a sequence of observed states and transaction issue events at each node and later comparing them will never will be able to distinguish the system from a simple centralized database with synchronous atomic transaction issue. This should be a sufficient level of consistency for every system.[28]

B. Extension to Full Consistency

To make ACC-1 systems be fully consistent, we must get ACC-1 involved in the process of reading a replica's state from the external world. We assume that we have a multi-node system with a tree topology, since that is the most general case (complete topologies are just simulations of tree topologies, so the same techniques will apply analogously). Further, it must be using the instantiation from §V-C and obeying the global primacy criterion from §VI-A.3.

The procedure to read the replica at node N is essentially to cause N to become the global primary node. This can be accomplished by introducing a new message called "primacy request" and proceding according to the following rules:

1) If N does not have global primacy, then it sends a primacy request message on the link where it does not have primacy.
2) Whenever a primacy request message is received at some M:
   a) If M has global primacy, then it relinquishes primacy on the link that sent it the primacy request message.
   b) Otherwise (i.e., if M does not have global primacy), once none of M's links are waiting for primacy,[29] it
      i) records that this message's sender is waiting for primacy and
      ii) forwards the primacy request message on the link where it does not have primacy.
3) Whenever a node receives primacy on a link (so that it becomes the global primary), it checks if one of its links is waiting for primacy. If so, it relinquishes primacy on that link.
4) Using the above, N will eventually become the global primary node. Once this happens, it observes the state of its replica.

[28]Note that one *can* find ways to distinguish this from a centralized system. For example, the real-world time of each observation and transaction issue event might imply a different interleaving of transactions from what the system produces, or that more transactions' results should have been visible in a particular observation.

[29]Note that other reception events need not be blocked while waiting for this condition, since the new message only serves to cause global primacy to be transfered, so its order of delivery with respect to other messages is not important.

We will call this the *observation procedure*. We already know that the global primary node's state is always the past state of every node as of the last message from it. Therefore, when global primacy is transfered, the state at the new global primary will be the past state of the previous global primary node as of the moment when it gave up that primacy.

Thus, since the transactions applied at the global primary are always the literal transaction that are issued in the system, we see that the observed state will necessarily be the last observed one (since it was observed at a global primary too) plus a sequential interleaving of the transactions that have since been issued and communicated to us. Therefore full consistency will hold.

This procedure can also be thought of as defining a checkpointing procedure. As it happens, an alternative implementation is possible that can also provide analogous semantics for systems with arbitrary definitions of Δ and that lack global primacy. It is in Appendix II. For providing full consistency, though, the procedure given in this section is better.

C. Application: Real-World Databases

In addition to providing full consistency, the observation procedure can also be used to let ACC-1 support real-world databases. So far we have support for read/write transactions that do not move information to the external world (call them "isolated transactions") and read-only transactions that *do* move information to the external world (call them "exporting transactions"), but thus far we have no support for read/write transactions that *also* move information to the external world (call them "general transactions").

Using the above, though, it is easy to support general transactions. Each general transaction can clearly be separated into two parts: an exporting transaction and an isolated transaction. The original general transaction is equivalent to doing those two parts in that order, provided that that is done atomically and that ACC-1 does not later re-order transactions so as to make that atomicity no longer respected by the final state.

Since we know that a global primary will always apply the literal received transactions, we can easily implement general transactions as follows: (i) become the global primary node using the observation procedure and atomically (ii) execute the exporting transaction part and (iii) issue the isolated transaction part. Received transactions will never later undo and redo the isolated transaction part, so the data read by the exporting transaction part will always be consistent with the state to which the isolated transaction is seen to have been applied.

We thus now have sufficient theory to use ACC-1 for real-world databases. By backing-up unrecoverably overwritten state, we can obtain a set of isolated transactions that satisfies the required group-theoretic properties and that can use the instantiation in §V-C, and every real transaction will be executable using either the issue of an isolated transaction, the observation procedure, or the procedure in the above paragraph.

Note however that the performance boost from using ACC-1 in a real-world database will be limited by the ratio of isolated transactions to general and exporting transactions, because the latter two types incur network delay when executed. In fact, in the worst case, ACC-1 may be considerably worse than a traditional concurrency controller, because we have not provided any way to have global primacy on a per-item basis, whereas traditional concurrency controllers do provide locks on a per-item basis. An extension to rectify this is probably possible, but beyond the scope of this paper.[30] Further, we have not developed any method to support multiple global primaries when they are doing read-only operations, whereas traditional concurrency controllers do offer shared locks. Again, this could probably be rectified.

Regardless, the ability of ACC-1 to increase performance in a real-world database may be severely limited, because isolated transactions will probably be the minority in most such systems. For example, in a SQL database, every SELECT statement issued by a user process is an exporting transaction. Only statements such as INSERT, UPDATE, and DELETE are isolated transactions—and only if they do not have the capability to return error codes, which in practice they would. Despite this, though, compound

[30]Note also that such an extension would probably re-introduce the possibility of distributed deadlock.

transactions built from those fundamental ones could still potentially be isolated transactions if they do not return their intermediate results (nor any other results) to the external world, but unless those are the majority then ACC-1 will not perform much better here than very simplistic distributed locking. However, it might be possible to salvage its usefulness by hybridizing it with a multiversion timestamp ordering algorithm. [5]

Alternatively, some real-world databases may not really need full consistency—they may be able to suffice with just sequential consistency. For example, consider a financial database that is used to provide the storage back-end for a bank's Internet banking website. Such systems are usually the motivation for designing what we think of intuitively as "fully" consistent databases, but observe that what the bank mainly cares about is just that, at the end of the day, money has been neither created nor destroyed. The operative phrase there is *at the end of the day*—that corresponds to just sequential consistency. The bank will probably not care if two users see two financial transactions occur in opposite order, so long as the final result is the same and money has been conserved. The users may not care much either.[31] Indeed, to some extent, financial systems already have these semantics, because many banking actions are not fully processed until one or more business days later—long after the user interface has said "transaction complete".

In addition to the above possibility of relaxed consistency, we will see another potential solution in §IX for systems where that is not an option.

VIII. Partial Replicas

We have so far assumed that every node has a complete replica of the shared state. In reality this is sometimes not desirable. Memory and disk space at nodes may prohibit complete replicas, or the overhead to set up a complete replica may be prohibitively high. Further, if a particular node is only working with a particular subset of the database's data, then its performance would be adversely affected by having to receive and process transactions for other data.

The solution of course is having partial replicas which only cache part of the database state. Extending ACC-1 to support this involves adding three new transactions: "open", "opened", and "close". We initially assume that there is a hierarchy of replicas which each contain at least everything below them, and the top-most replica contains everything in the database. Then:

- "Open" requests the current state of a currently uncached set of items to be sent to the sender; "opened" answers with the data.
- "Close" indicates that the sender has discarded its cache of a set of items.
- "Open" and "close" are sent upwards in the hierarchy, while "opened" is sent downwards.
- Every node maintains extra data for each of its downward links that records what data items are currently open at any of the nodes reachable from that link.
- When routing transactions, they are only sent on links that have the data item open that the transaction operates on (and the upward link, if any).[32]
- When an "open" transaction is received, it is answered and the receiving link's open data set is updated, provided that the data is available locally. Otherwise an open for the missing data is sent on the link to the higher level and the received one is answered once all the data is available.
- When a "close" transaction is received, the receiving link's open data set is updated.[33]
- The new transactions are synchronized as follows:

[31] There *are* some problematic cases, though. For example, if transactions get re-ordered and an account now has insufficient funds for a transaction that previously was seen to complete, then a user who saw only the intermediate state might want to get a notification about the change, say.

[32] If the transaction operates on multiple data items and a link has some of the output items open but not all the input items, then an alternate transaction must be sent which just writes the result to the open output items, and suitable implementations of Δ for it must be found.

[33] And if the receiver was waiting to close that thing itself, it now can.

- "Close" nullifies transactions that would have operated on the closed data and is itself unaffected by anything.[34]
- "Open" and "opened" do not affect anything and are also themselves unaffected.

We will not develop theory for this system,[35] but it has been well-tested by XCDE, which supports open/close at the path name granularity.

However, the situation is actually more complex in XCDE, because intermediate nodes do not maintain replicas (they are for routing only). As a result, open requests go right to the uppermost node, because intermediate ones never have the data to answer them, even when all that data is actually open by their other clients. This causes synchronization issues when the answer comes back: the "opened" transaction that contains the data for the newly opened item might end up being synchronized via Δ with a transaction that edits that item. The solution is to define Δ in that case to modify the data in "opened" in the same way that the transaction would modify the real data in the database.

Additionally, when Δ is extended in that way, a hierarchy of replicas that descend from a complete one is not necessary.[36] Instead, every node can maintain the open data set information for each of its links, and intermediate nodes can send a close message on any of their links whenever neither the intermediate node itself nor any of its other links is using a certain data item. "open" requests are then serviced either locally or by routing them to any link that has the requested data already open. Eventually they will get to a node that has the data and will answer the request, and the intervening nodes then route the answer back. This does however require that Δ then also be implemented for concurrent "open" and "close" requests to the same datum,[37] and also that additional logic is present to ensure that everything is always open by at least one node. The benefit though is that clients can receive close messages once they become the only node that has a certain thing open. This has the advantage that that client then need not send any messages for transactions on that item, which will be critical in §X.

These procedures can also be extended to systems that support dynamic creation/deletion of data items at the same granularity as open/close. One just potentially updates a link's open data set whenever a transaction is executed, and then adds a few more synchronization cases. XCDE does this, for example.

IX. ACC-1 Processes

Consider the case of a process at some ACC-1 node that repeatedly issues a general transaction (i.e., atomically observes the replica and applies a transaction to it). Each time, the transaction that it chooses to issue is a function of what it observes and its internal state, which is itself a function of everything that it has observed in the past. If the system has full consistency, then this process will not get any performance boost due to ACC-1, because it will have to become the global primary whenever it does anything. This corresponds well with the situation of a user process that performs ongoing operation on a SQL database, so it would be well worth optimizing this case.

Note that this process is not really moving any data out into the external world. It is only reading data to decide what transaction to issue next. One might expect then that it is possible to put this under the control of ACC-1 and improve performance, and indeed it is. We will refer to a process under the control of ACC-1 as an "ACC-1 process".

The approach is as follows:

[34]Transactions that operate partly on the closed data must be transformed so that the still-open data can be updated. This may be difficult, since the database state will not be available for reference. It may be necessary in such a system to define T' and κ to add the missing information.

[35]We conjecture that its validity can be verified by formulating "open", "opened", and "close" as normal transactions in an imaginary instantiation where each node stores and maintains *every* node's replica (and where thus the states truly are identical when the system quiesces) and then showing that the above behaves in the same way.

[36]Though XCDE is still architected hierarchically.

[37]The close simply nullifies the open. The sender of the open will later get the close and know to re-send the open on a different link.

1) Mark isolated transaction messages that came from such a process P with an indicator that says that they came from that ACC-1 process, and specify the set of items that P read into its internal state when it executed the corresponding general transaction.[38]
2) Use the instantiation in §V-C, except change it as follows.
   a) In the $b = true$ case (returning $B^{-1} * A * B$), if B is a transaction from some ACC-1 process P and records that P observed something that A would modify, then return just $B^{-1} * A$. Further, mark it with extra information so that it triggers the same case with any other transaction from P that it is synchronized with via Δ.
   b) In the $b = false$ case (returning A), if A is a transaction from some ACC-1 process P and records that P observed something that B would modify, then nullify it (i.e., return I). Further, trigger the same case if B is a transaction with the extra information that is added in case 2a.
3) Whenever any transactions from an ACC-1 process P are inverted on the local replica, restore the internal state of P to its state from before the oldest of the inverted transactions was performed.
4) If an ACC-1 process ever chooses based on its internal state to not execute a general transaction and instead either send something to the external world or exit, then force it to first observe the database using the procedure from §VII-B. Only perform the special operation if the internal state is not restored to a past state during the observation process.

The changes above to Δ satisfy the semisymmetry criterion—in the new case, both sides are A. Further, it can be seen that the result of synchronization will be some sequentially consistent interleaving of the transactions, except that each ACC-1 process P may be missing all of the transactions that it issued after some point.

However, the position of the remaining transactions from P in the interleaving will be consistent with the data that P read for each of those transactions. Additionally, because P's internal state will have been restored to what it was after the last such transaction, that internal state will be consistent with the database state. Thus the state of the system is equivalent to the hypothetical case where P simply has not yet issued any transactions after the last one visible in the result. Therefore, once ACC-1 processes continue with the issue of more transactions and eventually finish (after each successfully observing the database), the final state will be a sequentially consistent interleaving of a mutually possible sequence of transactions for each ACC-1 process (plus whatever other transactions are issued by normal processes).

The special option in rule 4 is to allow for processes that only *mostly* satisfy the ACC-1 process ideal, and which at other times do unrestorable things. Since it is based on the observation procedure which is already known to work, it clearly will be valid. In theory this would allow *any* process in a real-world database system to be turned into an ACC-1 process and get increased performance. However, the performance will still be limited by the frequency with which that process *really* transfers data to the external world, which may still be very often in a real-world database. Further, bringing an arbitrary existing such process under the control of ACC-1 may simply be an engineering infeasibility; it requires that the ACC-1-based database be able to track and revert the state of an external process that was never designed with that possibility in mind. That is unlikely to be feasible without some sort of supporting low-level operating system mechanism. As it happens, though, we are about to see examples of such mechanisms in the following two sections, which present our last two example applications.

X. Application: Distributed Shared Memory

Consider the task of solving some mathematical problem using a large computer network with a common address space. If the problem is easily parallelizable, then traditional concurrency control incurs little overhead and is thus a perfect solution. However, if the problem is not very parallelizable—perhaps due to chaotic data access patterns that are not feasible to predict or coordinate in advance—then concurrency control based on ACC-1 may allow for increased performance.

[38] Because we specify this set and use that information for synchronization, an ACC-1 process that observes the replica but does not change it must still issue some transaction. I comes to mind.

To formulate such a system, assume for simplicity that each processor at each computer is a fetch-and-execute register machine. The "database" is the contents of the common address space. Each processor has some internal state that consists of register contents, a program counter, etc.

Observe that each processor can be easily conceptualized as an ACC-1 process. On each "iteration" of the "process", it (1) observes the instruction pointed to by its current program counter and (2) executes that instruction. The procedure for executing an instruction will vary widely with the architecture, but the effect should be characterizable by (a) a potential change to the processor's registers (which in all interesting cases will include a change to the program counter) and (b) potentially some number of writes to memory.

To represent the effect of executing an instruction, we can create a single "write" transaction of the form "write data D to address A". Sequences of writes constructed with the $*$ operator suffice to represent multiple writes if instructions can have them. The transactions generated by the processors will additionally say that the issuing processor observed (1) the address equal to its program counter value and (2) any addresses that were read during execution of the instruction (e.g., input operands). The added ACC-1 process-related cases for Δ would only be triggered by writes to the same addresses that another transaction says some process read.

This situation conforms to the requirements of §IX, but it must be optimized to be practical. Currently every computer in the system would have to eventually execute every write that every other computer's processors do. That would prevent us from getting any performance boost at all, since every computer must do essentially everything that a single centralized computer would have to do.

To get a performance boost out of ACC-1, computers would have to use partial replicas based on the procedures in §VIII.[39] A reasonable granularity for open/close might be memory pages. Further, the non-hierarchical approach would have to be used so that clients would not need to send transactions for data that only they have open, because otherwise the top replica would still be doing everything that a single centralized computer would do, and there is also no way that the network could carry writes at the same rate that their issuing processor could execute them.

Assuming that operating on exclusively-owned pages is frequent enough so that the network is not overburdened with traffic, the above may be feasible to implement efficiently in real systems using virtual memory management techniques. Reads and writes to exclusively-owned pages could be executed natively on the hardware, while only reads and writes to shared pages would need to be interrupted and sent over the network. Further, whenever accessing a shared page, the processor's state would be recorded and all open pages would be lazily copied via the well-known copy-on-write technique.[40] That provides sufficient information to restore the previous state if a conflict occurs.

The system would also need a mechanism to dynamically decide when the benefit of keeping a shared page open (i.e., foregoing the cost of retrieving it once used) is not worth the cost (i.e., applying received writes to it) and thus close that page. ACC-1's benefit would then be in the twilight zone where a shared page is too frequently used by everyone for anyone to close it or get it exclusively. The benefit conferred would be the ability to access such a page without any network delay and yet get fully consistent semantics. The final performance boost would depend on many factors, including:

1) the network bandwidth relative to the rate of shared page accesses (so that the network is not overburdened),
2) the ratio of shared page accesses to exclusive page accesses (so that ACC-1 gets to save time frequently enough to make a difference),
3) the probability of a past state not having to be restored (so that the time spent on speculative execution pays off frequently enough),
4) the average number of distinct pages that are modified by a processor after a shared memory access (so that the cost of copying pages does not reduce the amount of real speculative execution that can

[39]Happily, this is an *easy* case of partial replicas, because there are no indivisible transactions that span data items.

[40]This is of course unnecessary on the machine that has global primacy.

be done to insignifant amounts), and of course
5) the speed of all the book-keeping.

Note that although this is best suited to distributed computation, any distributed program could use the above technique. However, since accessing a device is a transfer of information to the external world and must thus trigger observation as described in §IX, programs that interact with a human user might not receive any performance boosts. Note that storage devices such as hard drives could theoretically support restoring past state, so in that case one might bring them under the control of ACC-1 if they were accessed too frequently.

One could probably also adapt these techniques into a cache coherence protocol for centralized multi-processor systems, which would have an advantage over current such protocols in that the timing constraints would be more relaxed (because the communication would be asynchronous).

XI. Application: Autoparallelization

With computer systems becoming increasingly multi-processor (or multi-core), there is a desire to make it easier to develop multi-threaded programs which exploit the architecture of such systems. The ideal scenario would be to automatically transform serial programs into parallel ones. Asynchronous concurrency control techniques could potentially be adapted to do this.

Consider a procedure call in a program written in some procedural language, and suppose that the call does not directly return any data to the caller; i.e., its only effect is to modify the contents of the address space of the program. As a result, the calling code does not immediately need any results from the procedure, so the calling code could potentially continue in a new thread (the "speculative" thread) and execute in parallel with the procedure call (running in the "original" thread). However, the results will only be correct if the two threads' instructions are executed in an order that is equivalent to one in which the original thread completes entirely before the speculative thread runs; i.e., the calling code cannot be allowed to read or write anything that the original thread is going to modify, nor write anything that it is going to read (because it is precisely those orders that are not equivalent to the expected semantics of a procedure call).

It would certainly not be feasible to statically predict whether such violations will take place,[41] but there is another option; we can simply assume that there will be no violations, and then catch any that occur and correct for them. This can be modelled with ACC-1. Once we spawn off the speculative thread, we consider the two threads to be logically separate uniprocessor computers (with initially the same memory contents) that are using the distributed shared memory design from §X, but with two exceptions: (1) the original thread has primacy and it is not allowed to be exchanged and (2) ACC-1 messages from the speculative thread to the original thread are not sent.[42] Once the original thread completes the procedure call, that thread becomes defunct and the speculative thread continues as a conventional thread.

It can be seen that the final result in such a model will always have the semantics of a procedure call. Throughout execution, the memory contents of the original thread's logical computer are the same as they would be without autoparellization, and the original thread has global primacy so it can operate devices normally. Meanwhile, the speculative thread's memory contents are a *speculation* on what the address space would be if all code was executed serially up to the current point in the speculative thread; as the original thread executes, this speculation is updated, potentially reverting the speculative thread's work if a conflicting access is made. When the original thread completes, the speculation becomes correct.

To implement this model in the real world where both threads are really on one computer, one would have to use virtual memory techniques to track the actions of both threads. Each page would have to be potentially marked to indicate whether the speculative thread has (1) read from and/or (2) written to that page since it was launched (and in the case where it has modified the page, the original contents must be backed-up). Meanwhile, conflicting accesses by the original thread to the pages so marked would have

[41] Clearly it is at least as hard as the halting problem.

[42] This latter exception does not violate previous analysis, because it is the same as having an infinite network delay in that direction.

to be intercepted to trigger a restoration of the speculative thread and the pages it modified to a coherent snapshot from before the time of the earliest conflicting speculative access.

To truly make this mechanism automatic, it would also be necessary to automatically identify procedure calls that have a good chance of benefitting from parallelization. This could perhaps be done by trying it with every procedure call at first and then tracking the pay-off from each so that an intelligent decision can be made the next time. Unfortunately though, identifying and intercepting procedure calls may be difficult without some help from the program's compiler.

Three additional notes are worth making:

First, after speculative execution starts, it is possible to start it again with another procedure call in either the original or speculative thread. In the ACC-1 model, one simply expands the node for the thread that makes the call into two nodes, making the overall graph into a chain. This has two advantages. The obvious one is that, if there are more than two processors, then this allows them all to be used. However, it may also allow the system to "hedge its bets" regarding the pay-off from speculative execution. This is because, in the event that a roll-back is necessary, it is only the threads that have actually done conflicting accesses[43] that need be rolled-back. This may, therefore, increase the amount of useful speculative execution.

Second, this technique's usefulness is actually not limited to multi-processor systems. In a single-processor system, if a process blocks on an I/O operation (or any operation at all, including page swapping), the system could do speculative execution for the code after the procedure call that it is inside. Thus the computer would be able to get more out of its single CPU.

Third, in multi-core processors, some of this could potentially be implemented in hardware. In particular, that might considerably reduce the overhead in initiating speculative execution.

XII. Fault Tolerance

A common reason for turning a centralized system into a distributed one is to eliminate the single common point of failure that is inherent in a centralized system and instead get *no* common points of failure. However, if the distributed system is too tightly coupled, then the result can be actually *multiple* common points of failure, which is *worse*.

As presented thus far, distributed systems based on ACC-1 will suffer from this problem. In this section we consider a specific type of failure that ACC-1 is particularly weak against: a temporarily lost connection. That is, suppose that a 2-node ACC-1 link is operating normally and then the underlying network connection is lost and later re-established. What is necessary to ensure that the system continues correctly?

In essentially every network protocol, each node will have no way of knowing how many of its messages were processed by the other before the connection was lost (except of course that it knows that at least all the things that were acknowledged in the ACC-1 protocol have been processed). Therefore, some of the elements in its skew sequence may have already been applied at the other end. Further, there will not be any acknowledgements en route for them, because they would have all been lost when the connection went down, and the other end cannot re-send them because it does not know how many of them were already received.

As it happens though, there is a simple solution. Each node can count how many acknowledgements it has sent and how many it has received. Upon restoration of the connection, each node sends the sent acknowledgements count to the other. Once it receives the other's count, the difference between it and the local count of received acknowledgements is the number that were lost in the receiving direction when the connection went down. That node thus simulates receiving that many acknowledgements (i.e., removes that many elements from the beginning of its skew sequence). Further, all of the transactions that are left in its skew sequence correspond to ones that were lost in the sending direction when the connection went down, so it re-sends them all in order. It then sends any transactions that were issued after the connection was lost, whereafter it proceeds with normal operation.

[43] Or whose parents did before they were spawned; or who have accessed state that is going to be rolled back.

The simulated acknowledgement reception and the transactions re-send by the other node will combine to effectively restore the implicit state that ACC-1 "stores" in its en-route message streams. However, it will not be restored to precisely the same thing. All of the acknowledgements will be processed first, and all of the transactions will be processed last, regardless of the actual order in the message stream from before the connection loss. Further, the transactions will not be exactly the same; they will be whatever is in the skew sequence, not whatever was sent. However, these two differences cancel each other out. From Theorem 1 part (3), the elements in the skew sequence are the values that the opposite end would have produced for them after looping them each through the skew sequence that we believed it to have at the time. By processing all of the acknowledgements first, all of the elements in the opposite end's skew sequence that we knew about (and thus have already accounted for in the transactions that we re-send) will have been removed before any of those re-sent transactions are processed. Therefore precisely every transformation that would have been done had no connection loss occurred will still be done exactly once, so the net result will be the same.

Thus the above procedure will correctly recover from the loss of the connection and the users of ACC-1 will never see anything other than normal behaviour.

Note that if a connection is lost and re-established with a *different* node in a multi-node system (or re-established with the same one but without knowledge of the past skew sequence and counter values), then the above procedure does not apply. It would be possible in principle to adapt it to work in the case where the node that we previously were linked to is still contactable through the network, but if we cannot re-establish a direct link then the node itself has probably failed, so such a procedure would be of little use. In fact, in the case where our previous link partner has failed, there is no general-purpose way to transparently continue with normal operation. Instead, our replica state must be synchronized with that of our new link parter via some application-specific method. Admittedly this reveals a fundamental and unavoidable shortcoming of any asynchronous concurrency control system: there is no guarantee that completed transaction issue events will take affect on the database, because it is always possible that communication will be unrecoverably lost before they get there. Purely asynchronous concurrency control systems will, therefore, always lack the durability desideratum of transaction systems.

That may be less of a drawback than one would think. In a fully consistent system with a master server, that server will probably be the original global primary node and it will probably be given back global primacy after a node obtains and uses it. If so, the procedure for observing a replica's state will guarantee that all previous transactions have been applied to the master before the external world can find out that the corresponding transaction issue events completed. Thus, to the user, durability will be present. Note though that there is still no guarantee that the transactions will propagate to every single node in the network, because again it is possible that communication will be unrecoverably lost before that happens. However, most traditional concurrency control systems would not make that guarantee either.

Failure scenarios beyond those considered above (such as a failure of volatile memory) should not be substantially different from those in a traditional concurrency control system, so we forego analyzing them.

XIII. Conclusion

We have seen that ACC-1 provides a general framework for improving the performance of massively distributed systems—even when consistency cannot be sacrificed. Further, techniques inspired by asynchronous concurrency control may find use in a wide array of situations that are not directly related to either concurrency control or distributed systems. We have found that not all applications can benefit from ACC-1, and others would need to overcome significant engineering challenges to adopt it. Nonetheless, in the right situations, the potential pay-off could be significant.

APPENDIX I
PROOF OF THEOREM 1

We first need to determine exactly how the functions f_A and g behave. To express g, we first need to define a new function as follows:

Definition 14: $f_S : T' \times [T'] \to [T']$ is the function such that, for all $A' \in T'$ and $S \in [T']$, $f_A(A', S)$ is the final value of the skew sequence that the transaction reception event handler would produce if it received A' with its skew sequence initially equal to S.

Note that f_A and f_S are closely related. They are the results for the two different things that are computed in the same situation. Together they are essentially the denotational semantics of the transaction reception pseudo-code.

From the definitions and the transaction reception event handler pseudo-code, it is easy to see that these three functions are given as follows.

For f_A,

$$\begin{aligned} f_A(t, \{\}) &= t \\ f_A(t, \{t_1, t_2, ..., t_n\}) &= \delta(f_A(t, \{t_1, t_2, ..., t_{n-1}\}), t_n), n > 0 \end{aligned}$$

For f_S,

$$\begin{aligned} f_S(t, \{\}) &= \{\} \\ f_S(t, \{t_1, t_2, ..., t_n\}) &= \{\delta(t_1, t)\} + f_S(\delta(t, t_1), \{t_2, ..., t_n\}), n > 0 \end{aligned}$$

or, equivalently,

$$\begin{aligned} f_S(t, \{\}) &= \{\} \\ f_S(t, \{t_1, t_2, ..., t_n\}) &= f_S(t, \{t_1, t_2, ..., t_{n-1}\}) + \{\delta(t_n, f_A(t, \{t_1, t_2, ..., t_{n-1}\})\}, n > 0 \end{aligned}$$

And for g,

$$\begin{aligned} g(\{\}, L_2) &= L_2 \\ g(\{\alpha, t_2, ..., t_n\}, L_2) &= g(\{t_2, ..., t_n\}, (L_2)_{-1}) \\ g(\{t_1, t_2, ..., t_n\}, L_2) &= g(\{t_2, ..., t_n\}, f_S(t_1, L_2)), t_1 \neq \alpha, n > 0 \end{aligned}$$

or, equivalently,

$$\begin{aligned} g(\{\}, L_2) &= L_2 \\ g(\{t_1, t_2, ..., \alpha\}, L_2) &= g(\{t_1, ..., t_{n-1}\}, L_2)_{-1} \\ g(\{t_1, t_2, ..., t_n\}, L_2) &= f_S(t_n, g(\{t_1, ..., t_{n-1}\}, L_2)), t_n \neq \alpha, n > 0 \end{aligned}$$

We now state and prove a simple helper lemma:

Lemma 1: For all $s \in [T']$ and $A' \in T'$,

$$\kappa(f_A(A', s)) * \prod_{i=1}^{\#s} \kappa(\overline{f_S(A, s)}[i])^{-1} = (\prod_{i=1}^{\#s} \kappa(\overline{s}[i])^{-1}) * \kappa(A')$$

Proof: The proof is by induction on $\#s$.

Base case: $\#s = 0$

Then $s = \{\}$, so:

$$\kappa(f_A(A', s)) * \prod_{i=1}^{\#s} \kappa(\overline{f_S(A', s)}[i])^{-1} = (\prod_{i=1}^{\#s} \kappa(\overline{s}[i])^{-1}) * \kappa(A')$$
$$\Leftrightarrow \kappa(f_A(A', \{\})) * \prod_{i=1}^{0} \kappa(\overline{f_S(A', \{\})}[i])^{-1} = (\prod_{i=1}^{0} \kappa(\overline{\{\}}[i])^{-1}) * \kappa(A')$$
$$\Leftrightarrow \kappa(f_A(A', \{\})) = \kappa(A')$$
$$\Leftrightarrow \kappa(A') = \kappa(A')$$
$$\Leftrightarrow true \qquad Since\ LHS\ =\ RHS$$

So the equation holds for $\#s = 0$.
Inductive case: $\#s > 0$ and the equation holds for $\#s - 1$
Then $s_{\overline{-1}}$ is a sequence over the elements of T and has length of $s - 1$, so the inductive hypothesis tells us that:

$$\kappa(f_A(A', s_{\overline{-1}})) * \prod_{i=1}^{\#(s_{\overline{-1}})} \kappa(\overline{f_S(A', s_{\overline{-1}})}[i])^{-1} = (\prod_{i=1}^{\#(s_{\overline{-1}})} \kappa(\overline{s_{\overline{-1}}}[i])^{-1}) * \kappa(A') \tag{8}$$

Thus,

$$\begin{aligned}
&\kappa(f_A(A', s)) * \prod_{i=1}^{\#s} \kappa(\overline{f_S(A', s)}[i])^{-1} \\
&= \kappa(\delta(f_A(A', s_{\overline{-1}}), s[\#s])) * \prod_{i=1}^{\#s} \kappa(\overline{f_S(A', s)}[i])^{-1} \\
&= \kappa(\delta(f_A(A', s_{\overline{-1}}), s[\#s]))\times \\
&\quad \times \prod_{i=1}^{\#s} \kappa(\overline{f_S(A', s_{\overline{-1}}) + \{\delta(s[\#s], f_A(A', s_{\overline{-1}}))\}}[i])^{-1} \\
&= \kappa(\delta(f_A(A', s_{\overline{-1}}), s[\#s])) * \kappa(\delta(s[\#s], f_A(A', s_{\overline{-1}})))^{-1}\times \\
&\quad \times \prod_{i=1}^{\#(s_{\overline{-1}})} \kappa(\overline{f_S(A', s_{\overline{-1}})}[i])^{-1} \\
&= \kappa(\delta(f_A(A', s_{\overline{-1}}), s[\#s]))\times \\
&\quad \times (\kappa(f_A(A', s_{\overline{-1}}))^{-1} * \kappa(f_A(A', s_{\overline{-1}})) * \kappa(\delta(s[\#s], f_A(A', s_{\overline{-1}}))))^{-1}\times \\
&\quad \times \prod_{i=1}^{\#(s_{\overline{-1}})} \kappa(\overline{f_S(A', s_{\overline{-1}})}[i])^{-1} \\
&= \kappa(\delta(f_A(A', s_{\overline{-1}}), s[\#s]))\times \\
&\quad \times (\kappa(f_A(A', s_{\overline{-1}}))^{-1} * \kappa(s[\#s]) * \kappa(\delta(f_A(A', s_{\overline{-1}}), s[\#s])))^{-1}\times \\
&\quad \times \prod_{i=1}^{\#(s_{\overline{-1}})} \kappa(\overline{f_S(A', s_{\overline{-1}})}[i])^{-1} \qquad By\ (1) \\
&= \kappa(\delta(f_A(A', s_{\overline{-1}}), s[\#s]))\times \\
&\quad \times \kappa(\delta(f_A(A', s_{\overline{-1}}), s[\#s]))^{-1} * \kappa(s[\#s])^{-1} * (\kappa(f_A(A, s_{\overline{-1}}))^{-1})^{-1}\times
\end{aligned}$$

$$\times \prod_{i=1}^{\#(s_{\overline{-1}})} \kappa(\overline{f_S(A', s_{\overline{-1}})}[i])^{-1}$$

$$= \kappa(s[\#s])^{-1} * \kappa(f_A(A', s_{\overline{-1}})) * \prod_{i=1}^{\#(s_{\overline{-1}})} \kappa(\overline{f_S(A', s_{\overline{-1}})}[i])^{-1}$$

$$= \kappa(s[\#s])^{-1} * (\prod_{i=1}^{\#(s_{\overline{-1}})} \kappa(\overline{s_{\overline{-1}}}[i])^{-1}) * \kappa(A') \qquad By\ (8)$$

$$= (\prod_{i=1}^{\#(s_{\overline{-1}})+1} \kappa(\overline{s_{\overline{-1}} + \{s[\#s]\}}[i])^{-1}) * \kappa(A')$$

$$= (\prod_{i=1}^{\#s} \kappa(\overline{s}[i])^{-1}) * \kappa(A')$$

which is the equation's right-hand side.

Therefore, whenever the equation holds for $\#s - 1$, it also holds for $\#s$.

Therefore, by the principle of mathematical induction, the equation holds for all values of $\#s$ and Lemma 1 is true. ■

We can now proceed to prove Theorem 1.

Proof: The proof is by induction on e.

Base case: $e = 0$

Then up to the moment after event e, no event has occurred at N since the initialization of the system. Thus in particular no receive events have occurred, so $c_r(N, e) = 0$. Thus, since $n_t(\neg N, 0) = 0$, we have $k(N, e) = 0$. Further, after the zeroth event, the skew sequence is still empty and no transactions have been issued. Thus,

$$C(N, e) = B(\neg N, k(N, e))$$
$$\Leftrightarrow C(N, 0) = B(\neg N, 0)$$
$$\Leftrightarrow B(N, 0) * \prod_{i=1}^{L(N,0)} \kappa(\overline{S(N, 0)}[i])^{-1} = B(\neg N, 0)$$
$$\Leftrightarrow I * \prod_{i=1}^{0} \kappa(\{\}[i])^{-1} = I$$
$$\Leftrightarrow I = I$$
$$\Leftrightarrow true \qquad Since\ LHS\ =\ RHS$$

So (2) holds. Further, since no events have ever occurred, there in particular have been no transmit events, so $F(N, e)$ is empty. Thus it does indeed contain exactly $L(N, 0) = 0$ transactions and $L(\neg N, 0) = 0$ acknowledgements, so (3) is true. Further, since the range $[1, L(N, e)] = [1, 0]$ is empty, (4) is trivially true. Thus the statements are true for $e = 0$.

Inductive case: $e > 0$ and the statements are true for $e - 1$

Let E be the e^{th} event. Since $e > 0$, it is a normal event (as opposed to the "event" of initializing the system). There are exactly three types of normal events, so there are three cases for E:

1) E is a transaction issue event for some extended transaction A'
   From the properties of the local message passing, $B(N, e) = B(N, e - 1) * \kappa(A')$.
   From the event handler pseudo-code, $S(N, e) = S(N, e - 1) + \{A'\}$ (so $L(N, e) = L(N, e - 1) + 1$).

Since E is not a receive event, $k(N,e) = k(N,e-1)$. Thus,

$$
\begin{aligned}
& C(N,e) = B(\neg N, k(N,e)) \\
& \Leftrightarrow B(N,e) * \prod_{i=1}^{L(N,e)} \kappa(\overline{S(N,e)}[i])^{-1} \\
& = B(\neg N, k(N,e)) \\
& \Leftrightarrow B(N,e-1) * \kappa(A') * \prod_{i=1}^{L(N,e-1)+1} \kappa((\{A'\} + \overline{S(N,e-1)})[i])^{-1} \\
& = B(\neg N, k(N,e-1)) \\
& \Leftrightarrow B(N,e-1) * \kappa(A') * \kappa(A')^{-1} * \prod_{i=1}^{L(N,e-1)} \kappa(\overline{S(N,e-1)}[i])^{-1} \\
& = B(\neg N, k(N,e-1)) \\
& \Leftrightarrow B(N,e-1) * I * \prod_{i=1}^{L(N,e-1)} \kappa(\overline{S(N,e-1)}[i])^{-1} \\
& = B(\neg N, k(N,e-1)) \\
& \Leftrightarrow B(N,e-1) * \prod_{i=1}^{L(N,e-1)} \kappa(\overline{S(N,e-1)}[i])^{-1} \\
& = B(\neg N, k(N,e-1)) \\
& \Leftrightarrow C(N,e-1) = B(\neg N, k(N,e-1)) \\
& \Leftrightarrow true \qquad\qquad By\ inductive\ hypothesis
\end{aligned}
$$

Thus, (2) holds.
From the pseudo-code and the fact that $k(N,e) = k(N,e-1)$, we have that $F(N,e) = F(N,e-1) + \{A'\}$.
Thus, since $F(N,e-1)$ contains exactly $L(N,e-1)$ extended transactions by the inductive hypothesis, $F(N,e) = F(N,e-1) + \{A\}$ contains exactly $L(N,e-1) + 1 = L(N,e)$.
Further, since $L(\neg N, k(N,e)) = L(\neg N, k(N,e-1))$ (because $k(N,e) = k(N,e-1)$) and $F(N,e-1)$ contains exactly $L(\neg N, k(N,e-1))$ acknowledgements, so too does $F(N,e) = F(N,e-1) + \{A\}$ contain exactly $L(\neg N, k(N,e-1)) = L(\neg N, k(N,e))$.
Thus (3) holds.
Now, since $F(N,e) = F(N,e-1) + \{A'\}$ and $S(N,e) = S(N,e-1) + \{A'\}$, we have that, for all $i \in [1, \#F(N,e-1)]$, $F(N,e)[i] = F(N,e-1)[i]$, and for all $i \in [1, L(N,e)]$, $S(N,e)[i] = S(N,e-1)[i]$.
Also, since $k(N,e) = k(N,e-1)$, we have that $S(\neg N, k(N,e)))) = S(\neg N, k(N,e-1))))$.
Therefore, (4) for $i \leq L(N,e-1) = L(N,e) - 1$ follows directly from the inductive hypothesis, because none of the terms in it have changed.
As for the case of $i = L(N,e)$, since $F(N,e-1)$ contains exactly $L(N,e-1)$ transactions by the inductive hypothesis, the $L(N,e)^{th}$ transaction in $F(N,e) = F(N,e-1) + \{A'\}$ is A; i.e., $F(N,e)[p(N,e,i)] = A'$.
Further, this implies that $F(N,e)_{1..(p(N,e,i)-1)} = F(N,e-1)$, which contains exactly $L(\neg N, k(N,e-1)) = L(\neg N, k(N,e))$ acknowledgements by the inductive hypothesis.
From the definition of g, it can thus be seen that

$$g(F(N,e)_{1..(p(N,e,i)-1)}, S(\neg N, k(N,e))) = \{\}$$

because the function will remove a total of $L(\neg N, k(N,e))$ elements from $S(\neg N, k(N,e))$ during its transformations of it, and that is precisely the number that are in it. Therefore,

$$\begin{aligned}
&S(N,e)[i] = f_A(F(N,e)[p(N,e,i)], X) && (i = L(N,e)) \\
&\quad where\ X = g(F(N,e)_{1..(p(N,e,i)-1)}, S(\neg N, k(N,e))) \\
&\Leftrightarrow A' = f_A(A', \{\}) \\
&\Leftrightarrow A' = A' \\
&\Leftrightarrow true && Since\ LHS\ =\ RHS
\end{aligned}$$

Thus (4) holds for $i = L(N,e)$ too, and thus holds for all $i \in [1, L(N,e)]$. Thus the statements are true.

2) E is an acknowledgement reception event
Then the $k(N,e)^{th}$ event at $\neg N$ must have been a transaction reception event, since no other kind of event sends an acknowledgement.
Thus $F(N, e-1)$ must contain at least one transaction, since otherwise $\neg N$ could not have received anything to acknowledge.
Thus since $F(N, e-1)$ contains exactly $L(N, e-1)$ transactions by the inductive hypothesis, it must be that $L(N, e-1) > 0$.
Therefore the semantics of the event handler's removal of the first element of `S` are well-defined, and we have that $B(N,e) = B(N, e-1)$ and $S(N,e) = S(N, e-1)_{-1}$ (so $L(N,e) = L(N, e-1) - 1$).
Since E is a receive event, $k(N,e) > k(N, e-1)$.
By the definition of $F(N, e-1)$ and the in-order property of the communication channel, the extended transaction that was received by $\neg N$ as the $k(N,e)^{th}$ event must have been the earliest one in $F(N, e-1)$: i.e., $F(N, e-1)[p(N, e-1, 1)]$.
For the same reasons, any other messages in $F(N, e-1)$ before index $p(N, e-1, 1)$ (which will all be acknowledgements) must have been received between event numbers $k(N, e-1)$ and $k(N,e)$. Therefore, $F(N,e) = F(N, e-1)_{-p(N,e-1,1)}$.
Additionally, because acknowledgement reception events are the only events that do not send a message, all events at $\neg N$ with numbers in the range $(k(N, e-1), k(N,e))$ must have been acknowledgement reception events. Further, since during that time exactly $p(N, e-1, 1) - 1$ acknowledgements were received there, it must be that $k(N,e) = k(N, e-1) + p(N, e-1, 1)$.
Thus, from the acknowledgement reception event handler's pseudo-code, it must be that $S(\neg N, k(N,e) - 1) = S(\neg N, k(N, e-1))_{-(p(N,e-1,1)-1)}$ (so $L(\neg N, k(N,e)-1) = L(\neg N, k(N, e-1)) - (p(N, e-1, 1) - 1)$) and $B(\neg N, k(N,e) - 1) = B(\neg N, k(N, e-1))$.
Since we have established that event $k(N,e)$ at $\neg N$ must have been a transaction reception event, we have from the transaction reception event handler's pseudo-code that

$$\begin{aligned}
B(\neg N, k(N,e)) = B(\neg N, k(N,e) - 1) \times \\
\times \kappa(f_A(F(N, e-1)[p(N, e-1, 1)], S(\neg N, k(N,e) - 1)))
\end{aligned}$$

and

$$S(\neg N, k(N,e)) = f_S(F(N, e-1)[p(N, e-1, 1)], S(\neg N, k(N,e) - 1))$$

$$(so\ L(\neg N, k(N,e)) = L(\neg N, k(N,e)-1))$$

Thus from the above two equations and the previous observation,

$$\begin{aligned} B(\neg N, k(N,e)) &= B(\neg N, k(N,e-1))\times \\ &\quad \times \kappa(f_A(F(N,e-1)[p(N,e-1,1)], S(\neg N, k(N,e-1))_{-(p(N,e-1,1)-1)})) \end{aligned}$$

and

$$\begin{aligned} S(\neg N, k(N,e)) &= f_S(F(N,e-1)[p(N,e-1,1)], S(\neg N, k(N,e-1))_{-(p(N,e-1,1)-1)}) \\ &(so\ L(\neg N, k(N,e)) = L(\neg N, k(N,e-1)) - (p(N,e-1,1)-1)) \end{aligned}$$

Therefore,

$$\begin{aligned} &C(N,e) = B(\neg N, k(N,e)) \\ &\Leftrightarrow B(N,e) * \prod_{i=1}^{L(N,e)} \kappa(\overline{S(N,e)}[i])^{-1} \\ &= B(\neg N, k(N,e)) \\ &\Leftrightarrow B(N,e-1) * \prod_{i=1}^{L(N,e-1)-1} \kappa(\overline{S(N,e-1)_{-1}}[i])^{-1} \\ &= B(\neg N, k(N,e-1)) * \kappa(f_A(F(N,e-1)[p(N,e-1,1)], S(\neg N, k(N,e-1))_{-(p(N,e-1,1)-1)})) \\ &\Leftrightarrow (B(N,e-1) * \prod_{i=1}^{L(N,e-1)} \kappa(\overline{S(N,e-1)}[i])^{-1}) * \kappa(S(N,e-1)[1]) \\ &= B(\neg N, k(N,e-1)) * \kappa(f_A(F(N,e-1)[p(N,e-1,1)], S(\neg N, k(N,e-1))_{-(p(N,e-1,1)-1)})) \\ &\Leftrightarrow C(N,e-1) * \kappa(S(N,e-1)[1]) \\ &= B(\neg N, k(N,e-1)) * \kappa(f_A(F(N,e-1)[p(N,e-1,1)], S(\neg N, k(N,e-1))_{-(p(N,e-1,1)-1)})) \end{aligned}$$

Now, since we determined that the first $p(N,e-1,1)-1$ elements of $F(N,e-1)$ are acknowledgements, we have from the definition of g that

$$g(F(N,e-1)_{1..(p(N,e-1,1)-1)}, S(\neg N, k(N,e-1))) = S(\neg N, k(N,e-1))_{-(p(N,e-1,1)-1)}$$

Further, from the inductive hypothesis we know that

$$\begin{aligned} S(N,e-1)[1] &= f_A(F(N,e-1)[p(N,e-1,1)], X) \\ &where\ X = g(F(N,e-1)_{1..(p(N,e-1,1)-1)}, S(\neg N, k(N,e-1))) \end{aligned}$$

So $S(N,e-1)[1] = f_A(F(N,e-1)[p(N,e-1,1)], S(\neg N, k(N,e-1))_{-(p(N,e-1,1)-1)})$, which when reversed allows us to finish the above as follows,

$$\begin{aligned} &\Leftrightarrow C(N,e-1) * \kappa(S(N,e-1)[1]) \\ &= B(\neg N, k(N,e-1)) * \kappa(S(N,e-1)[1]) \\ &\Leftrightarrow C(N,e-1) \\ &= B(\neg N, k(N,e-1)) \\ &\Leftrightarrow true \qquad\qquad By\ inductive\ hypothesis \end{aligned}$$

So (2) holds.
From the work above, we already know that $F(N,e)$ contains exactly $p(N,e-1,1)-1$ fewer acknowledgements than $F(N,e-1)$—just as $S(\neg N,k(N,e))$ contains the same fewer elements than $S(\neg N,k(N,e-1))$—and that $F(N,e)$ also contains exactly one fewer transaction that $F(N,e-1)$—just as $S(N,e)$ contains one fewer element than $S(N,e-1)$—so (3) follows from the inductive hypothesis.
As for (4), we have from the inductive hypothesis that:

$$\begin{aligned} S(N,e-1)[i] &= f_A(F(N,e-1)[p(N,e-1,i)],X) \\ &\quad where\ X = g(F(N,e-1)_{1..(p(N,e-1,i)-1)}, S(\neg N,k(N,e-1))) \end{aligned}$$

Since the first $p(N,e-1,1)-1$ elements of $F(N,e-1)$ are acknowledgements, it can be seen from the definition of g and the work above that:

$$\begin{aligned} X &= g((F(N,e-1)_{-(p(N,e-1,1)-1)})_{1..(p(N,e-1,i)-1)-(p(N,e-1,1)-1)}, Y) \\ &\quad where\ Y = S(\neg N,k(N,e-1))_{-(p(N,e-1,1)-1)} \end{aligned}$$

Consider the case of $i \in [1, L(N,e-1))$ and rewrite the above by substituting $i+1$ for i.

$$S(N,e-1)[i+1] = f_A(F(N,e-1)[p(N,e-1,i+1)],X)$$

$$\begin{aligned} where\ X &= g((F(N,e-1)_{-(p(N,e-1,1)-1)})_{1..(p(N,e-1,i+1)-1)-(p(N,e-1,1)-1)}, Y) \\ and\ Y &= S(\neg N,k(N,e-1))_{-(p(N,e-1,1)-1)} \end{aligned}$$

The sequence $(F(N,e-1)_{-(p(N,e-1,1)-1)})_{1..(p(N,e-1,i+1)-1)-(p(N,e-1,1)-1)}$ is always non-empty and its first element is always a transaction, so:

$$\begin{aligned} &g((F(N,e-1)_{-(p(N,e-1,1)-1)})_{1..(p(N,e-1,i+1)-1)-(p(N,e-1,1)-1)}, Y) \\ &\quad where\ Y = S(\neg N,k(N,e-1))_{-(p(N,e-1,1)-1)} \\ &= g(((F(N,e-1)_{-(p(N,e-1,1)-1)})_{1..(p(N,e-1,i+1)-1)-(p(N,e-1,1)-1)})_{-1}, Y) \\ &\quad where\ Y = f_S(F(N,e-1)_{-(p(N,e-1,1)-1)}[1], S(\neg N,k(N,e-1))_{-(p(N,e-1,1)-1)}) \\ &= g((F(N,e-1)_{-p(N,e-1,1)})_{1..(p(N,e-1,i+1)-1)-(p(N,e-1,1)-1)-1}, Y) \\ &\quad where\ Y = f_S(F(N,e-1)[p(N,e-1,1)], S(\neg N,k(N,e-1))_{-(p(N,e-1,1)-1)}) \\ &= g(F(N,e)_{1..(p(N,e-1,i+1)-1)-(p(N,e-1,1)-1)-1}, S(\neg N,k(N,e)) \\ &= g(F(N,e)_{1..(p(N,e-1,i+1)-1)-p(N,e-1,1)}, S(\neg N,k(N,e)) \\ &= g(F(N,e)_{1..(p(N,e,i)-1)}, S(\neg N,k(N,e)) \end{aligned}$$

And thus:

$$
\begin{aligned}
S(N,e-1)[i+1] &= f_A(F(N,e-1)[p(N,e-1,i+1)],X) \\
&\quad where\ X = g(F(N,e)_{1..(p(N,e,i)-1)}, S(\neg N,k(N,e))) \\
&= f_A(F(N,e-1)_{-p(N,e-1,1)}[p(N,e-1,i+1)-p(N,e-1,1)],X) \\
&\quad where\ X = g(F(N,e)_{1..(p(N,e,i)-1)}, S(\neg N,k(N,e))) \\
&= f_A(F(N,e)[p(N,e-1,i+1)-p(N,e-1,1)],X) \\
&\quad where\ X = g(F(N,e)_{1..(p(N,e,i)-1)}, S(\neg N,k(N,e))) \\
&= f_A(F(N,e)[p(N,e,i)],X) \\
&\quad where\ X = g(F(N,e)_{1..(p(N,e,i)-1)}, S(\neg N,k(N,e)))
\end{aligned}
$$

Further, since $S(N,e) = S(N,e-1)_{-1}$, we have that $S(N,e-1)[i+1] = S(N,e)[i]$. Hence,

$$
\begin{aligned}
S(N,e)[i] &= f_A(F(N,e)[p(N,e,i)],X) \\
&\quad where\ X = g(F(N,e)_{1..(p(N,e,i)-1)}, S(\neg N,k(N,e))
\end{aligned}
$$

which is (4). Thus the statements are true.

3) E is a transaction reception event for some extended transaction A'
Then from the local message passing properties and the event handler's pseudo-code, $B(N,e) = B(N,e-1) * \kappa(f_A(A, S(N,e-1)))$ and $S(N,e) = f_S(A', S(N,e-1))$ (so $L(N,e) = L(N,e-1)$).
This is a receive event, so $k(N,e) > k(N,e-1)$.
Further, the $k(N,e)^{th}$ event at $\neg N$ must have been a transaction issue event for A', because no other event can trigger sending A'.
Thus, from the transaction issue event handler pseudo-code,

$$B(\neg N, k(N,e)) = B(\neg N, k(N,e)-1) * \kappa(A')$$

and

$$S(\neg N, k(N,e)) = S(\neg N, k(N,e)-1) + \{A'\}$$

$$(so\ L(\neg N, k(N,e)) = L(\neg N, k(N,e)-1) + 1)$$

Also, all events at $\neg N$ with numbers in the range $(k(N,e-1), k(N,e))$ must have been acknowledgement reception events, since that is the only event that does not send a message.
Define $x = k(N,e) - k(N,e-1) - 1$. It is the number of such events that occurred.
x can be at most $p(N,e-1,1)-1$, because—by definition of p—$F(N,e-1)[1..p(N,e-1,1)-1]$ is the largest starting subsequence of $F(N,e-1)$ that contains only acknowledgements.
Since acknowledgement reception events do not change the replica state, we have that $B(\neg N, k(N,e)-1) = B(\neg N, k(N,e-1))$.
Thus, $B(\neg N, k(N,e)) = B(\neg N, k(N,e-1)) * \kappa(A')$.
Further, from the above and the event handler's pseudo-code, we have that $F(N,e) = F(N,e-1)_{-x} + \{\alpha\}$.
We further have from the acknowledgement reception event handler's pseudo-code that $S(\neg N, k(N,e)-1) = S(\neg N, k(N,e-1))_{-x}$, so $S(\neg N, k(N,e)) = S(\neg N, k(N,e-1))_{-x} + \{A'\}$ (and so $L(\neg N, k(N,e)) = L(\neg N, k(N,e-1)) - x + 1$).
So:

$$
\begin{aligned}
& C(N,e) = B(\neg N, k(N,e)) \\
& \Leftrightarrow B(N,e) * \prod_{i=1}^{L(N,e)} \kappa(\overline{S(N,e)}[i])^{-1} \\
& \quad = B(\neg N, k(N,e)) \\
& \Leftrightarrow B(N,e-1) * \kappa(f_A(A', S(N,e-1))) \times \\
& \qquad \times \prod_{i=1}^{L(N,e-1)} \kappa(\overline{f_S(A', S(N,e-1))}[i])^{-1} \\
& \quad = B(\neg N, k(N,e-1)) * \kappa(A') \\
& \Leftrightarrow B(N,e-1) * (\prod_{i=1}^{L(N,e-1)} \kappa(\overline{S(N,e-1)}[i])^{-1}) * \kappa(A') \\
& \quad = B(\neg N, k(N,e-1)) * \kappa(A') && \textit{By Lemma 1} \\
& \Leftrightarrow B(N,e-1) * \prod_{i=1}^{L(N,e-1)} \kappa(\overline{S(N,e-1)}[i])^{-1} \\
& \quad = B(\neg N, k(N,e-1)) \\
& \Leftrightarrow C(N,e-1) = B(\neg N, k(N,e-1)) \\
& \Leftrightarrow true && \textit{By inductive hypothesis}
\end{aligned}
$$

Thus (2) holds.
From the inductive hypothesis, $F(N, e-1)$ contains exactly $L(N, e-1)$ transactions and $L(\neg N, k(N, e-1))$ acknowledgements, and we know that $F(N,e) = F(N,e-1)_{-x} + \{\alpha\}$, so $F(N,e)$ contains exactly $L(N,e-1) = L(N,e)$ transactions and $L(\neg N, k(N,e-1)) - x + 1 = L(\neg N, k(N,e))$ acknowledgements. Thus (3) is true.
As for (4), first note that $p(N,e,i) = p(N,e-1,i) - x$ for all valid i. Then from that and the above,

$$
\begin{aligned}
& S(N,e)[i] = f_A(F(N,e)[p(N,e,i)], X) \\
& \quad where\ X = g(F(N,e)_{1..(p(N,e,i)-1)}, S(\neg N, k(N,e))) \\
& \Leftrightarrow f_S(A', S(N,e-1))[i] = f_A(W,X) \\
& \quad where \\
& \qquad W = (F(N,e-1)_{-x} + \{\alpha\})[p(N,e-1,i) - x] \\
& \qquad X = g((F(N,e-1)_{-x} + \{\alpha\})_{1..(p(N,e-1,i)-x-1)}, Y) \\
& \qquad Y = S(\neg N, k(N,e-1))_{-x} + \{A'\} \\
& \Leftrightarrow f_S(A', S(N,e-1))[i] = f_A(W,X) \\
& \quad where \\
& \qquad W = (F(N,e-1) + \{\alpha\})[p(N,e-1,i)] \\
& \qquad X = g(((F(N,e-1) + \{\alpha\})_{-x})_{1..(p(N,e-1,i)-x-1)}, Y) \\
& \qquad Y = (S(\neg N, k(N,e-1)) + \{A'\})_{-x} \\
& \Leftrightarrow f_S(A', S(N,e-1))[i] = f_A(W,X) \\
& \quad where
\end{aligned}
$$

$$
\begin{aligned}
&W = (F(N, e-1) + \{\alpha\})[p(N, e-1, i)] \\
&X = g(((F(N, e-1) + \{\alpha\})_{1..(p(N,e-1,i)-1)})_{-x}, Y) \\
&Y = (S(\neg N, k(N, e-1)) + \{A'\})_{-x} \\
\Leftrightarrow\ & f_S(A', S(N, e-1))[i] = f_A(W, X) \\
& \quad where \\
&\quad W = F(N, e-1)[p(N, e-1, i)] \\
&\quad X = g((F(N, e-1)_{1..(p(N,e-1,i)-1)})_{-x}, Y) \\
&\quad Y = (S(\neg N, k(N, e-1)) + \{A'\})_{-x} \qquad By\ definition\ of\ p
\end{aligned}
$$

Now observe that since the first x elements of $F(N, e-1)_{1..(p(N,e-1,i)-1)}$ are all α, $g(F(N, e-1)_{1..(p(N,e-1,i)-1)}, s) = g((F(N, e-1)_{1..(p(N,e-1,i)-1)})_{-x}, s_{-x})$ for any s (by definition of g). Thus, continuing the above by applying this equation in reverse, we have that

$$
\begin{aligned}
\Leftrightarrow\ & f_S(A', S(N, e-1))[i] = f_A(W, X) \\
& \quad where \\
&\quad W = F(N, e-1)[p(N, e-1, i)] \\
&\quad X = g(F(N, e-1)_{1..(p(N,e-1,i)-1)}, Y) \\
&\quad Y = S(\neg N, k(N, e-1)) + \{A'\} \\
\Leftrightarrow\ & \delta(S(N, e-1)[i], f_A(A', S(N, e-1)_{1..i-1})) = f_A(W, X) \\
& \quad where \\
&\quad W = F(N, e-1)[p(N, e-1, i)] \\
&\quad X = g(F(N, e-1)_{1..(p(N,e-1,i)-1)}, Y) \\
&\quad Y = S(\neg N, k(N, e-1)) + \{A'\} \qquad By\ definition\ of\ f_S
\end{aligned}
$$

To help prove the last line above, we now prove the following nested lemma.
Lemma 2: For all $i \in [1, L(N, e-1)]$,

$$
\begin{aligned}
&g(F(N, e-1)_{1..(p(N,e-1,i)-1)}, S(\neg N, k(N, e-1)) + \{A'\}) \\
&= g(F(N, e-1)_{1..(p(N,e-1,i)-1)}, S(\neg N, k(N, e-1))) + U \\
&\quad where\ U = \{f_A(A', S(N, e-1)_{1..i-1})\}
\end{aligned} \tag{9}
$$

Proof: The proof is by induction on i. Note that we are thus now doing nested induction.
Base case: $i = 1$

$$
\begin{aligned}
&g(F(N, e-1)_{1..(p(N,e-1,i)-1)}, S(\neg N, k(N, e-1)) + \{A'\}) \\
&= g(F(N, e-1)_{1..(p(N,e-1,i)-1)}, S(\neg N, k(N, e-1))) \\
&+ \{f_A(A', S(N, e-1)_{1..i-1})\} \\
&\Leftrightarrow g(F(N, e-1)_{1..(p(N,e-1,1)-1)}, S(\neg N, k(N, e-1)) + \{A'\}) \\
&= g(F(N, e-1)_{1..(p(N,e-1,1)-1)}, S(\neg N, k(N, e-1))) \\
&+ \{f_A(A', S(N, e-1)_{1..0})\} \\
&\Leftrightarrow g(F(N, e-1)_{1..(p(N,e-1,1)-1)}, S(\neg N, k(N, e-1)) + \{A'\}) \\
&= g(F(N, e-1)_{1..(p(N,e-1,1)-1)}, S(\neg N, k(N, e-1))) \\
&+ \{f_A(A', \{\})\}
\end{aligned}
$$

$\Leftrightarrow g(F(N,e-1)_{1..(p(N,e-1,1)-1)}, S(\neg N, k(N,e-1)) + \{A'\})$
$= g(F(N,e-1)_{1..(p(N,e-1,1)-1)}, S(\neg N, k(N,e-1)))$
$+ \{A'\}$
$\Leftrightarrow (S(\neg N, k(N,e-1)) + \{A'\})_{-(p(N,e-1,1)-1)}$
$= S(\neg N, k(N,e-1))_{-(p(N,e-1,1)-1)} + \{A'\}$ *Since the first $(p(N,e-1,1)-1)$ elements of $F(N,e-1)$ are α by definition*

$\Leftrightarrow S(\neg N, k(N,e-1))_{-(p(N,e-1,1)-1)} + \{A'\}$
$= S(\neg N, k(N,e-1))_{-(p(N,e-1,1)-1)} + \{A'\}$
$\Leftrightarrow true$ *Since LHS = RHS*

Therefore (9) holds for $i = 1$.
Inductive case: $i > 1$ and (9) holds for $i-1$
(We distinguish here between the two active inductive hypotheses with the terms "outer" and "inner".)
Then $i-1$ is in the range $[0, L(N,e-1)]$.
$F(N,e-1)[p(N,e-1,i-1)]$ is a transaction and every element in

$$F(N,e-1)_{(p(N,e-1,i-1)+1)..(p(N,e-1,i)-1)}$$

is α (by definition of p).
Thus,

$g(F(N,e-1)_{1..(p(N,e-1,i)-1)}, S(\neg N, k(N,e-1)) + \{A'\})$
$= f_S(F(N,e-1)[p(N,e-1,i-1)], X)_{-(p(N,e-1,i)-1-p(N,e-1,i-1))}$
where
$X = g(F(N,e-1)_{1..(p(N,e-1,i-1)-1)}, Y)$
$Y = S(\neg N, k(N,e-1)) + \{A'\}$ *By the above and the definition of g*

$= f_S(F(N,e-1)[p(N,e-1,i-1)], X)_{-(p(N,e-1,i)-1-p(N,e-1,i-1))}$
where
$X = g(F(N,e-1)_{1..(p(N,e-1,i-1)-1)}, Y)$
$+ \{f_A(A', S(N,e-1)_{1..(i-1)-1})\}$
$Y = S(\neg N, k(N,e-1))$ *By inner inductive hypothesis*

$= (f_S(F(N,e-1)[p(N,e-1,i-1)], X) + U)_{-z}$
where
$z = p(N,e-1,i) - 1 - p(N,e-1,i-1)$
$X = g(F(N,e-1)_{1..(p(N,e-1,i-1)-1)}, S(\neg N, k(N,e-1)))$
$U = \{\delta(f_A(A', S(N,e-1)_{1..(i-1)-1}), V)\}$
$V = f_A(F(N,e-1)[p(N,e-1,i-1)], X)$ *By definition of f_S*

$$
\begin{aligned}
&= (f_S(F(N,e-1)[p(N,e-1,i-1)],X)+U)_{-z} \\
&\quad where \\
&\qquad z = p(N,e-1,i)-1-p(N,e-1,i-1) \\
&\qquad X = g(F(N,e-1)_{1..(p(N,e-1,i-1)-1)}, S(\neg N, k(N,e-1))) \\
&\qquad U = \{\delta(f_A(A', S(N,e-1)_{1..(i-1)-1}), S(N,e-1)[i-1])\} \qquad By\ outer\ inductive\ hypothesis \\
&= (f_S(F(N,e-1)[p(N,e-1,i-1)],X)+U)_{-z} \\
&\quad where \\
&\qquad z = p(N,e-1,i)-1-p(N,e-1,i-1) \\
&\qquad X = g(F(N,e-1)_{1..(p(N,e-1,i-1)-1)}, S(\neg N, k(N,e-1))) \\
&\qquad U = \{f_A(A', S(N,e-1)_{1..i-1})\} \qquad By\ definition\ of\ f_A \\
&= f_S(F(N,e-1)[p(N,e-1,i-1)],X)_{-z}+U \\
&\quad where \\
&\qquad z = p(N,e-1,i)-1-p(N,e-1,i-1) \\
&\qquad X = g(F(N,e-1)_{1..(p(N,e-1,i-1)-1)}, S(\neg N, k(N,e-1))) \\
&\qquad U = \{f_A(A', S(N,e-1)_{1..i-1})\} \\
&= g(F(N,e-1)_{1..(p(N,e-1,i)-1)}, S(\neg N, k(N,e-1)))+U \\
&\quad where\ U = \{f_A(A', S(N,e-1)_{1..i-1})\} \qquad By\ definition\ of\ g
\end{aligned}
$$

which is the right-hand side of (9). Thus, whenever it holds for $i-1$, it also holds for i. Therefore, by the principle of mathematical induction, (9) holds for all i, so Lemma 2 is true. ■

Now, continuing where we left off,

$$
\begin{aligned}
&S(N,e)[i] = f_A(F(N,e)[p(N,e,i)],X) \\
&\quad where\ X = g(F(N,e)_{1..(p(N,e,i)-1)}, S(\neg N, k(N,e))) \\
&\Leftrightarrow \delta(S(N,e-1)[i], f_A(A', S(N,e-1)_{1..i-1})) = f_A(W,X) \\
&\quad where \\
&\qquad W = F(N,e-1)[p(N,e-1,i)] \\
&\qquad X = g(F(N,e-1)_{1..(p(N,e-1,i)-1)}, Y) \\
&\qquad Y = S(\neg N, k(N,e-1)) + \{A'\} \\
&\Leftrightarrow \delta(S(N,e-1)[i], f_A(A', S(N,e-1)_{1..i-1})) = f_A(W,X) \\
&\quad where \\
&\qquad W = F(N,e-1)[p(N,e-1,i)] \\
&\qquad X = g(F(N,e-1)_{1..(p(N,e-1,i)-1)}, Y)+U \\
&\qquad Y = S(\neg N, k(N,e-1)) \\
&\qquad U = \{f_A(A', S(N,e-1)_{1..i-1})\} \qquad By\ Lemma\ 2 \\
&\Leftrightarrow \delta(S(N,e-1)[i], U') = \delta(f_A(W,X), U') \\
&\quad where \\
&\qquad W = F(N,e-1)[p(N,e-1,i)] \\
&\qquad X = g(F(N,e-1)_{1..(p(N,e-1,i)-1)}, S(\neg N, k(N,e-1)))
\end{aligned}
$$

$$
\begin{aligned}
& U' = f_A(A', S(N, e-1)_{1..i-1}) && By\ definition\ of\ f_A \\
\Leftrightarrow\ & \delta(S(N, e-1)[i], U') = \delta(S(N, e-1)[i], U') && \\
& where\ U' = f_A(A', S(N, e-1)_{1..i-1}) && By\ inductive\ hypothesis \\
\Leftrightarrow\ & true && Since\ LHS\ =\ RHS
\end{aligned}
$$

Thus equation (4) holds, so the statements are true.

Therefore, whenever the statements hold for $e-1$, they also hold for e, regardless of the type of the e^{th} event.

Therefore, by the principle of mathematical induction, the statements hold for all e and Theorem 1 is true. ■

APPENDIX II
ALTERNATE OBSERVATION PROCEDURE

We introduce four new messages: "observe", "done", "clear", and "capture". They carry no data. Then, whenever a user process at node N wants to observe the replica, the system proceeds according to the following rules:

1) Node N broadcasts an "observe" message on every link.
2) Everywhere that an observe message is received, it is forwarded on all other links.
3) Every node that sees an observe message blocks transaction issue events (including N).
4) Whenever an observe message reaches a node that has no other links, a "done" message is sent back.
5) Once each intermediate node receives a done message for each of the observe messages that it sent, it sends a done message back to the sender of its own received observe message.
6) Once N receives all of its done messages like in (5), the user process reads the state of the replica.
7) After that, node N broadcasts a "clear" message on every link.
8) Everywhere that a clear message is received, it is forwarded on all other links.
9) Every node that sees a clear message unblocks transaction issue events (including N).

Further, concurrent observations are governed by the following additional rules:

1) If a user process at some node M wants to observe the replica and transaction issue events are currently blocked due to rule 3 above, then that user process simply waits until M receives its clear message and then immediately reads the state of the replica.
2) If a node is waiting for a done message on a link and instead sees an observe message,[44] then:
   a) If the receiving node has primacy on that link, it ignores the received observe message.
   b) Otherwise (i.e., if it does not have primacy),[45] it transforms the received observe message into a "capture" message.
3) Whenever a node sees a capture message (including the one that generates it above):
   a) It marks the link that received it (or the one that received the observe message that caused it) as the source of a past received observe message (i.e., the link on which we should send a done message once we get one from the other links).
   b) Further, if that node already had a link marked thusly, then it clears that setting (so that we now expect to receive a done message from it) and forwards the capture message on that link.

In the simple case of no concurrent observations, it is fairly easy to see that, once all the done messages have reached N, there will be no transaction messages en route to it anywhere in the system. Further, no new transaction messages will be sent towards N at any node M until M has received its clear message from N. Lastly, any transactions en route towards M from N will reach it before that.

[44]This is due to two concurrent observations where neither started during the block period of the other.

[45]The choice of which primacy case causes capture is arbitrary.

As a result of the above and Theorem 2, the nodes' replica states will all be equal at the moments in time respectively when they each process their clear message. Additionally, any transactions issued at a node after that will reach other nodes after all acknowledgements or transaction messages for the transactions visible in the observation, and therefore they will not trigger synchronization via Δ for any such transactions.

Therefore, all future states of the system will be equal to the observed state plus whatever ACC-1 produces for the transactions that are issued after that. Thus this procedure guarantees that all nodes will agree that what N observed is a historical state of the system.

As for concurrent observations, case 1 is quite simple to analyze; we already know that the replica states are all equal when the clear message is processed, so the concurrent observations will observe the same thing. Case 2 is more complex, but it can be seen that essentially what happens is that a "capture" message is generated at the point in the network where the concurrent observations collide, and that message flows back along the path of one of those observations and turns it into a branch of the other. Thus after that it is the same as in case 1.

Note though that the handling of case 1 (and case 2 once reduced to it) introduces a potential drawback in that the observed state might be missing some transactions that have so far been issued (in the real-world-time sense). This can be easily rectified if desired by changing the behaviour so that nodes restart their observation after the clear instead of observing when it is received.

ACKNOWLEDGMENT

The author would like to thank Larry Chen, Andrew Craik, and Tom Levesque for their valuable work in co-developing XCDE, without which ACC-1 would never have been discovered.

Further thanks go to Andrew Craik for helping to proof-read this paper.

REFERENCES

[1] J. J. Rotman, *An Introduction to the Theory of Groups, 4th ed.*, New York: Springer-Verlag, 1995.

[2] C. Sun and C. Ellis, "Operational Transformation in Real-Time Group Editors: Issues, Algorithms, and Achievements," in *Proc. of the ACM 1998 conference on Computer Supported Cooperative Work (CSCW'98)*, Seattle, Washington, USA, Nov. 1998, pp. 59-68.

[3] L. Lamport, "How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs," *IEEE Trans. Computers*, vol. 28, no. 9, pp. 690-691, Sept. 1979.

[4] X. Defago, A. Schiper, and P. Urban. Totally ordered broadcast and multicast algorithms: A comprehensive survey. TR DSC/2000/036, EPFL, Switzerland, Sept. 2000.

[5] D. P. Reed. Naming and Synchronization in a Decentralized Computer System. TR MIT-LCS-TR-205, Massachusetts Institute of Technology, Cambridge, Massachusetts, Sept. 1978.

Tristan Schmelcher graduated from the University of Waterloo in 2007 with a Bachelor of Applied Science in Computer Engineering. He currently resides in Vancouver, BC pending a move to Seattle, WA to start work at Google.