◆ The Inherent Difficulty of Timely Primary-Backup Replication

Pramod Koppol, Kedar S. Namjoshi, Thanos Stathopoulos, and Gordon T. Wilfong

We show that existing methods for primary-backup replication may disrupt the timing behavior of an underlying service to the extent of making it unusable. Furthermore, we prove that this problem is inherent to the standard primary-backup model. The formal proof is based on an analysis of the "local knowledge" available to each party in a correct primary-backup protocol. This negative result implies that entirely new approaches are needed to resolve the problem; on the positive side, the proof offers some hints for designing a solution. © 2012 Alcatel-Lucent.

Introduction

The advent of cloud computing is changing the way in which computing services are offered. It is a challenge to offer latency- and jitter-sensitive services in a cloud environment (e.g., for telecommunications). Since users also expect high availability for these services, a central question is whether it is possible to combine time sensitivity (low delay and jitter) with high network performance and high availability.

We examine a widely-used class of protocols for fault tolerance, which provide high availability using a pair of machines: a primary machine with an associated backup, as shown in **Figure 1**. It is desirable that the primary and backup machines be placed as far away from one another as possible in order to minimize the risk of a catastrophic service failure. All existing protocols for primary-backup replication, however, require regular roundtrip synchronization between the two machines. This imposes a lower bound of a roundtrip delay on network processing. Given the requirement for low-latency, the lower bound limits the extent to which the primary and backup machines can be separated. This increases the possibility of a joint failure, and therefore reduces the extent of the fault tolerance guarantees that can be provided.

A natural question is whether new primary-backup protocols can be devised which are not subject to the roundtrip delay bound. We prove that this is *impossible*: the problem is inherent in the primary-backup model. The proof abstracts from the (unknown) protocols by reasoning solely in terms of the local state of knowledge of the various parties in the system. We show that in every correct protocol, packets can be released by the primary to the environment only after the primary "knows" that the backup is aware of the latest state changes. Hence, in normal operation, the backup must continuously seek to increase its knowledge regarding the state of the primary. A beautiful theorem from Chandy and Misra

Bell Labs Technical Journal 17(2), 15–24 (2012) © 2012 Alcatel-Lucent. • DOI: 10.1002/bltj.21541

[3] connects every gain in knowledge to a causal chain of messages, showing that the roundtrip message exchange is unavoidable.

Primary-Backup Protocols

A primary-backup protocol allows the state of a server, which can be thought of as a state machine, to be recovered after the failure of a machine on which the server is executed. We consider protection for a single crash failure. In *active replication* protocols [8, 9], all replicas receive the same messages in the same order. This suffices for fault tolerance if message processing is deterministic. *Passive replication* requires periodic checks on the server state; thus after a primary failure, recovery proceeds with the backup server starting from the latest checkpoint state. Implementations may also adopt hybrid approaches.

Panel 1. Abbreviations, Acronyms, and Terms

FF—Fault-free
FT—Fault-tolerant
TCP—Transmission Control Protocol

In both approaches, the root cause of timing disruptions is the synchronization between primary and backup. We examine more closely the source of the disruptions arising in Remus [4], which is a representative implementation of passive replication. An analysis of active replication provides similar conclusions. A simplified view of the Remus synchronization is given below, and shown in **Figure 2**.

1. Primary and backup are synchronized at the start of an *epoch* (the period between checkpoints).

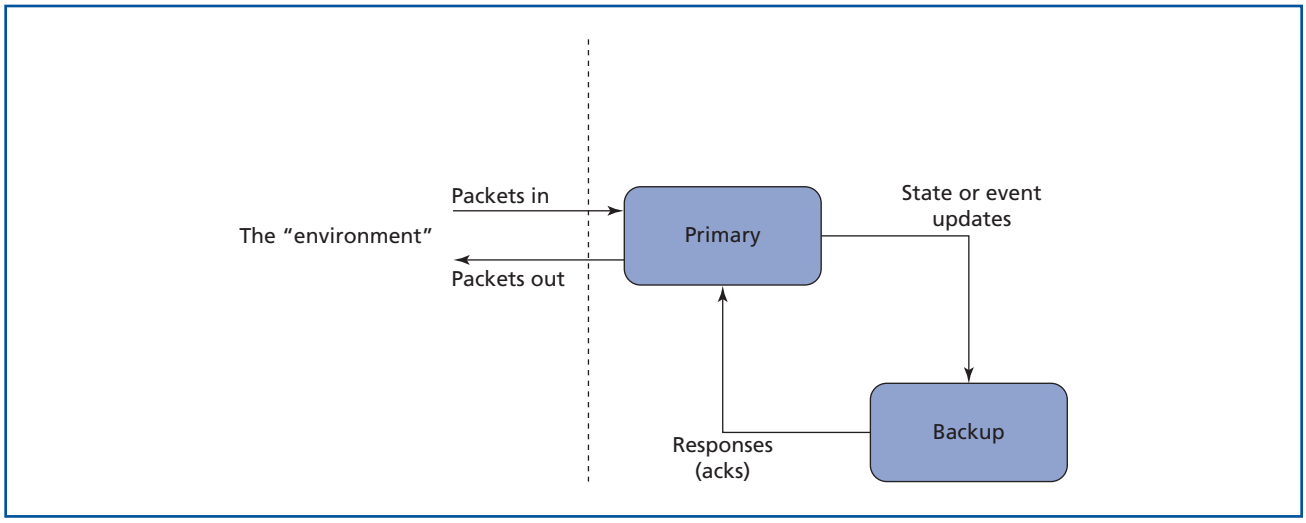


Figure 1.
Primary-backup replication.

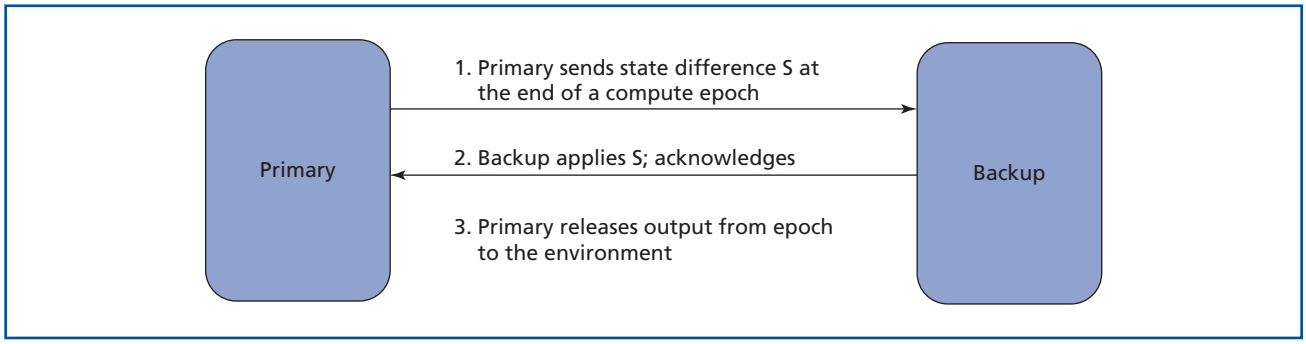


Figure 2.
Passive (state-based) replication.

2. Primary receives and processes a sequence of input messages; the output messages generated are buffered.
3. Primary sends its current state to the backup and waits for an acknowledgment.
4. Primary receives acknowledgment from backup; synchronization is complete.
5. Primary sends the buffered output to the environment and starts a new epoch.

Buffering is required in Step 2, since releasing output messages immediately can lead to an inconsistent view of the primary state between the backup and the environment after a primary failure. This is a special case of the theorem we show in this paper. Let T represent the roundtrip time between the primary and the backup. Suppose that, without any synchronization, the primary is able to process input arriving at a constant rate $r > 1/T$. At rate r, at least one message arrives during synchronization (Step 3 and Step 4). Since the primary is suspended at Step 3, these messages must be processed in the next epoch (after Step 4). If there is insufficient slack time during that epoch, some messages must remain unprocessed. In repeating this sequence of events, unprocessed messages accumulate beyond the bounds of the input buffers, which results in an unbounded delay in processing input messages.

A possible resolution is to drop some of the input messages, but that may result in a service failure (e.g., for voice packets), or a reduction in throughput. The Remus system resolves the problem by allowing the primary to process input while waiting for an acknowledgement from the backup. Output produced in this phase, which is concurrent with the primary-backup synchronization, is kept in a second output buffer, which is not released at Step 5, but instead forms the output buffer for the next epoch.

While the two-buffer solution resolves the problem of an unbounded delay in processing inputs, the roundtrip delay between primary and backup, which is incurred between the generation of an output packet and its eventual release to the environment, still remains. Unless the delay is small, it can severely impact both interactive services (e.g., a voice conference call) as well as reliable data transfer (TCP throughput is inversely proportional to delay). On the other hand, the small delay requirement hampers the flexible deployment of services, since it implies that either the primary and backup must be placed physically close together, or that there should be a high-bandwidth, low-latency link between them. A natural question is whether there are alternative—possibly more complex—primary-backup protocols which can overcome this problem. We show that this is not the case: *the problem is inherent to the primary-backup organization*.

Another problem with the protocol described above is that releasing output all at once (Step 5) can cause bursty traffic. In an experiment with Remus on a single audio stream, we observed that burstiness caused by Step 5, and delay due to checkpointing, can seriously degrade audio quality. Burstiness is a simpler problem, which can be fixed by appropriately smoothing-out the output.

Formal Analysis

We show that timing disruptions are inevitable for any primary-backup mechanism where there is no direct communication between the backup and the environment until the primary fails. The computing model is that of multiple processes communicating over a message-passing network. No assumptions are made about the network: it may lose, reorder, or duplicate messages. It is assumed that processes fail by halting, and that the failure can be detected—this is the well-known *fail-stop* model. The central claim is stated in the following theorem.

Theorem: For any generic primary-backup mechanism, there exists a service and an environment for which timing disruption is inevitable during fault-free operation.

We provide a sketch of the key argument before giving the full proof. The main difficulty for the proof is that it must quantify over *all* correct primary-backup mechanisms. This is done through an analysis of the states of knowledge of the relevant parties: the environment (E), the primary (P), the backup (B) and the service (S). Knowledge is represented by an assertion, "M *knows* b," which holds for a process M and predicate b in a computation x if b holds at *all* computations y which agree with x on the history of events for M. The knowledge predicate is essentially a

way of saying that the local view of M in computation x is *consistent* with the predicate b being true. Knowledge is thus a fundamental concept in the analysis of distributed protocols, where the evolution of a process must depend only on its local view of the global network state.

We show that a key invariant must be true of any correct primary-backup protocol: for any computation x and a visible predicate b on the state of S, if E *knows* b at x, then E *knows* (B *knows* not(E knows not(b))) at x. Roughly speaking, a predicate on the service state is visible if it is possible for the environment to distinguish, by means of message exchanges, whether the service is in a state satisfying that predicate. From this invariant, we show that any increase of knowledge by E—precisely, if E *knows* not(b) holds at x and E *knows* b holds at an extension y of x—forces a causal chain of messages going through the processes in the order <P;B;P;E> in the interval (x,y). This follows from [3], which connects knowledge gain to the existence of causal chains. The chain <P;B;P> is a roundtrip synchronization between primary and backup, which must occur before a message received by E increases its knowledge of the state of S.

The preceding reasoning applies to any environment and service. For some combinations, there may not be any visible predicates. We show how to choose a particular E and S which has a computation which can be partitioned into infinitely many disjoint intervals, in each of which E gains new knowledge. Based on the previous results, a synchronizing process chain is required for each of those intervals. This induces the behavior analyzed in the previous section; hence, timing disruption is inevitable.

In the following, we describe the computation model, provide a formal definition of knowledge, and give the detailed proof.

Basic Definitions

Processes may be thought of as state machines communicating via point-to-point messages. The communication network can be lossy, and may reorder or duplicate messages. Processes execute according to interleaving semantics: i.e., at every point in time, a specific process is chosen to execute an action. We take the basic definitions of computation, knowledge, process chain, and local predicates from Chandy and Misra's paper [3]. We summarize the key definitions here for completeness; the Chandy-Misra paper should be consulted for full detail.

A *computation* is a sequence of *events*. An event is either an internal process transition, or a "send" or "receive" for a message. This defines a "happens-before" relationship between events as in [8]. A computation is required to be downward-closed according to the happens-before order (i.e., every receive must have a corresponding send).

For computations x and y, $x \leq y$ means that x is a prefix of y (y is an extension of x). A *predicate* is a statement that has a truth value on computations, and which is insensitive to the ordering of concurrent events. For computations x and y, and process P, x[P]y means that P has the same sequence of events in x and y. For a predicate q, a computation x, and process P, we say that P *knows* q *at* x if, for all computations y such that x[P]y, the property q holds of y (informally, q holds in all computations consistent with P's history in x). A predicate q is *local* to process P if for all computations x, one of P *knows* q or P *knows* not(q) holds at x (informally, q is fully determined by the local history of P). There is a *process chain* $<P_1; P_2; \ldots ; P_n>$ in a computation x if there is a sequence of events e in x such that for each k, e[k] is an event of P_k, and successive events are causally ordered in x (i.e., there is a sequence of events ordered by the happens-before relation from e[k] to e[k+1] for all k: $1 \leq k < n$).

To illustrate the definitions, consider **Figure 3**. This shows a single computation formed from the history of two concurrent processes, A and B. Each dashed line represents a consistent state of the system; it can be viewed as defining the set of all events which precede the line. Each set is closed under the happens-before relation: i.e., if an event y belongs to the set, and event x happens-before y, then x must also be in the set. In state S_0, process A cannot know if the value of x in B is 0, as B may not have received the message yet. This is also true at state S_1 and S_2, since given process A's history, those states are indistinguishable to it from S_0. On the other hand, at state S_3, process A has received an acknowledgement from process B; thus, all states consistent with this history

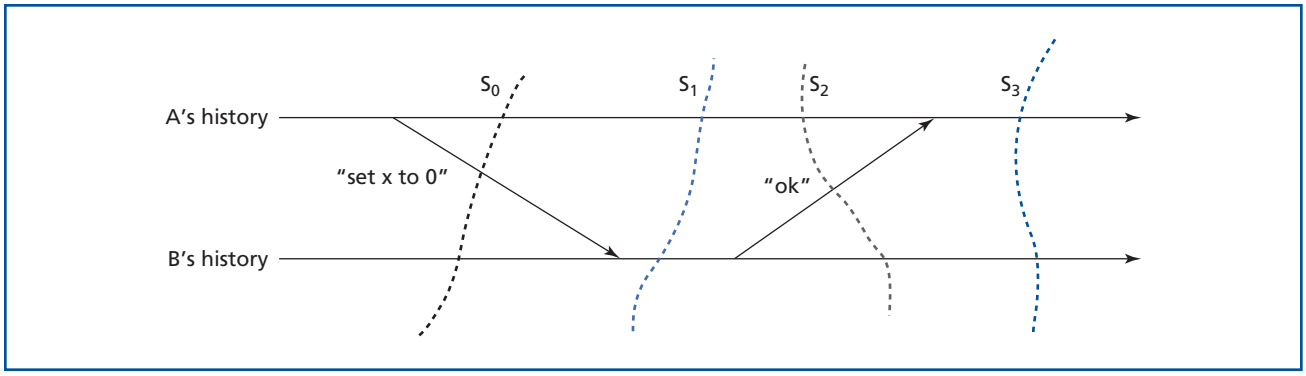


Figure 3.
Knowledge and message passing.

must have the value 0 for x at B (this assumes, of course, that B does not change x on its own).

Modeling Computation and Fault Tolerance

In order to define fault tolerance in general terms, we first define a reference system—one without any failures—and a system which tolerates failures. The fault-free reference system FF has two processes: an environment (E) and a service (S). The fault-tolerant system FT has three processes: the environment E, the primary (P) and the backup (B).

In order to relate the computations of FT and FF, we suppose that there is a way to map states of P and B to states of S. The proper mapping at any point on a computation of FT is determined by the active process: if the primary is active, its state is the one chosen to map to S; if the primary has failed, the state of B is mapped to S. Thus, for any computation of FT, there is an induced sequence made up of events at E and mappings of the state of the active process to S.

A first requirement is that of *safety:* FT must not produce computations which cannot belong to FF. A second requirement is that of *completeness*: any computation of FF must be reproduced in a fault-free way by FT. We take care of both by requiring that FF and the fault-free portion of FT are *bisimular upto stuttering* (i.e., finite repetition) [1]. Bisimularity matches-up the events of E, as well as the states of S. (A state s of FT and state t of FF match if the state of S that s is mapped to equals the state of S in t.)

Bisimularity is a standard notion for comparing two programs: it is a form of back-and-forth equivalence. A bisimulation B is a symmetric relation on states such that for any s,t related by B: 1) s and t satisfy the same atomic propositions (in our case, they agree on the state of S), and 2) for any transition labeled 'a' from s to state u, there is a transition labeled by 'a' from t to v such that u and v are related by B (in our case, labels are events of E). Taking stuttering into account results in a more complex definition, but the essence remains the same: a pair of related states have the same observable future behavior.

We define *fault tolerance* as follows. For every fault-free computation there is the possibility of a fault at the end of the computation. Faults are *fail-stop*: the state of the processes involved do not change, the failing process halts in its current state. (We may suppose that the state after a failure can be distinguished from the state before a failure by a special flag, since all process states are unchanged.) Let s be the state reached by a primary failure at the end of computation x. Fault tolerance is defined by the requirement that s is stuttering-bisimular to the state reached at the end of some fault-free computation y, which the environment cannot distinguish from x: i.e., a computation y such that y is fault-free and y[E]x holds. The definition ensures that the continuation of events as observed by the environment after a fault is one of the potential continuations which would have been

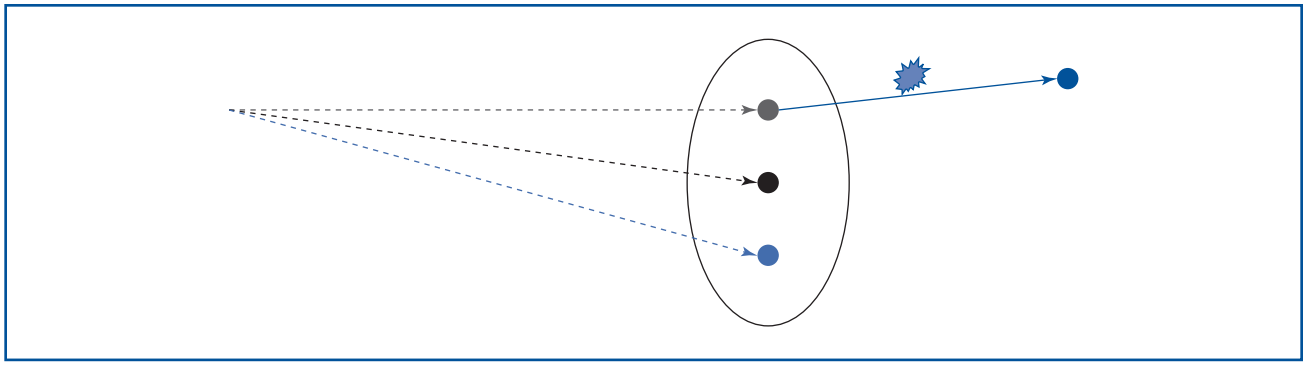


Figure 4.
Defining fault tolerance.

observable before the fault. This definition is illustrated in **Figure 4**. The dashed lines indicate that from the initial state, as far as the environment can tell, the system could be in any of the three states in the marked region. On a fault from the gray state, fault tolerance requires only that the system transit to a state that is similar to one of the states in the marked region; in this case, the blue state.

Finally, there is an important restriction on the *structure* of a standard primary-backup system. In a fault-free computation of FT, there is no direct communication between environment E and backup B; after a primary failure, all communications are between E and B.

The Proof

Definition: A predicate q on the state of S is *visible* in FF if, for any pair of reachable states (e_1, s_1) and (e_2, s_2) of FF such that s_1 satisfies q but s_2 does not satisfy q (e_1 and e_2 are the corresponding environment states) there is a *query* (a non-state changing message interaction) which can be performed from (e_1, s_1) and (e_2, s_2) but with differing results. It is required that the result of a query is functional, i.e., deterministic.

In the following, we restrict the scope of quantification in the "knows" operator to be that of failure-free computations of FT. Only a primary failure is considered in the proof.

Theorem 1: Let q be a visible predicate on the state of S. For any failure-free computation x of FT, if E knows q at x, then E knows B knows (not (E knows not(q))) at x.

Proof: (The term "not(E *knows* not(q))"—the negation dual of *knows*—simplifies to the following. It holds of a computation x if there is a computation y indistinguishable to E from x such that q holds at y.)

The interpretation of q on a failure-free computation is given by the final state of P, mapped to S. Hence, q is a local predicate of P for failure-free computations.

The proof is by contradiction. Consider a computation x where the hypothesis E *knows* q holds, but the conclusion does not. By the definition of knowledge, there is a fault-free computation y such that x[E]y holds but B *knows* (not (E *knows* not(q))) is false at y. Hence, there is a fault-free computation z such that y[B]z and (E *knows* not(q)) at z.

As x[E]y, E *knows* q holds at y. Informally, we have two failure-free computations y and z such that B is "confused" between the two: the local history of B is the same in both, but the predicate q is true at y but false at z. In fact, a stronger statement can be made: as E *knows* q at y, the predicate q is true on *all* failure-free computations that are indistinguishable from y to the environment; and as E *knows* not(q) at z, the predicate q is false on *all* failure-free computations that are indistinguishable from z to the environment. **Figure 5** illustrates this scenario.

Now consider the possibility of a primary failure from both y and z. Let s_0 be the state reached after failure from y, and s_1 the state reached after failure from z. By the definition of fault tolerance, s_0 is bisimular to the state t_0 reached after some failure-free computation, say y', that is indistinguishable to the environment from y. Similarly, s_1 is equivalent to state

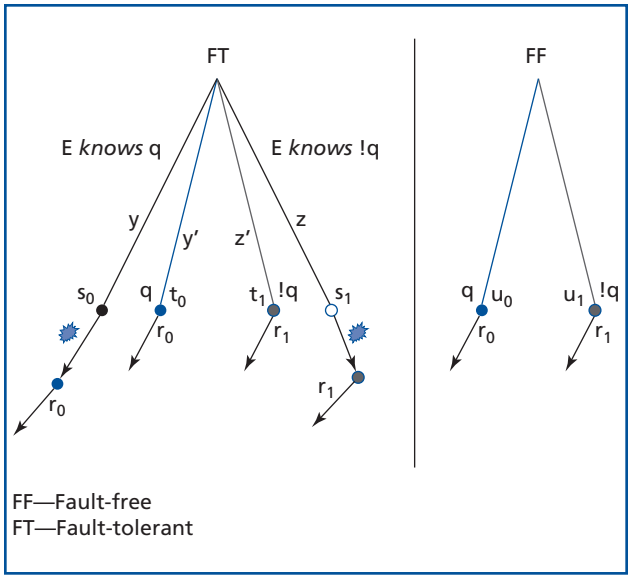


Figure 5.
Illustrating the proof of Theorem 1: states with the same color are bisimular; negation is abbreviated as "!".

t_1 reached by a failure-free computation, say z′, that is indistinguishable to the environment from z.

By bisimularity, y′ and z′ must have similar computations in FF, ending in states u_0 and u_1 respectively. As q is false of all computations which are indistinguishable to the environment from y, the predicate q must hold of y′. Similarly, q must be false of z′. By bisimularity, q must hold for u_0 and fail to hold of u_1. As q is a visible predicate, there is a query m which is possible from these states but which results in differing responses, say r_0 and r_1. By the completeness of bisimularity, the query m must be possible from y′ and z′ as well, and must result in the same responses. By bisimularity within FT, m must be possible from s_0 and s_1 and result in the same responses. However, as y and z have the same history for B, and the responses from s_0 and s_1 are generated by the backup, the result of processing the query at the backup after the failure is non-deterministic, a contradiction to the assumption of deterministic query responses. **Endproof**

Note 1: It is worthwhile to point out that the proof makes no assumption about how the internal structure of P and B relates to that of the service S. The only constraints (safety, completeness) relate behavior, not structure. For instance, P may be a state machine obtained by arbitrarily transforming the state machine for S, subject to the behavior constraints. We also do not assume full determinism of state machines; the only assumption is that queries have a deterministic response.

Note 2: The stronger claim "E *knows* q implies E *knows* B *knows* q" holds under a stricter definition of fault tolerance, where the state after failure has to be bisimular to the state before failure.

Theorem 1 makes a statement about any fault-tolerant implementation of a service. We use this to derive the result necessitating a round trip synchronization for a standard primary-backup configuration. This derivation makes crucial use of the structural requirement on standard configurations.

Theorem 2: Consider a standard primary-backup system. Let q be a visible predicate on the state of S. Let computations x and y be fault-free computations such that $x \leq y$ and E knows not(q) at x, while E knows q at y. Then there is a process chain <P;B;P;E> in the interval (x, y).

Proof. As E *knows* not(q) at x, the assertion B *knows* (not(E *knows* not(q))) is false at x. (If it were true, it would imply that not(E *knows* not(q)) holds at x, which contradicts the assumption.) As E *knows* q at y, by Theorem 1, E *knows* B *knows* (not(E *knows* not(q))) holds at y. By the knowledge gain theorem ([3], Theorem 5), there is a process chain <B; E> in the interval (x, y). By the structural assumption, any chain from B to E in a fault-free computation must pass through P: hence, the chain is really a chain <B;P;E> in the interval (x,y).

Next, we show that this chain can be extended on the left to a chain <P;B;P;E> in (x,y). The proof is by contradiction. Consider the *last* receive event by the process E in the interval (x,y); call it e. All subsequent events for process E are send and local events. By ([3], Lemma 4]), those events do not lead to knowledge gain; so that as E *knows* B *knows* (not(E *knows* not(q))) holds at y, the same assertion holds for the downward closure of event e according to the happens-before relation. We refer to the downward closure as computation z; the chain <B;P;E> in (x,y) must be included in (x,z), so z is a strict extension of x. The situation is illustrated in **Figure 6**.

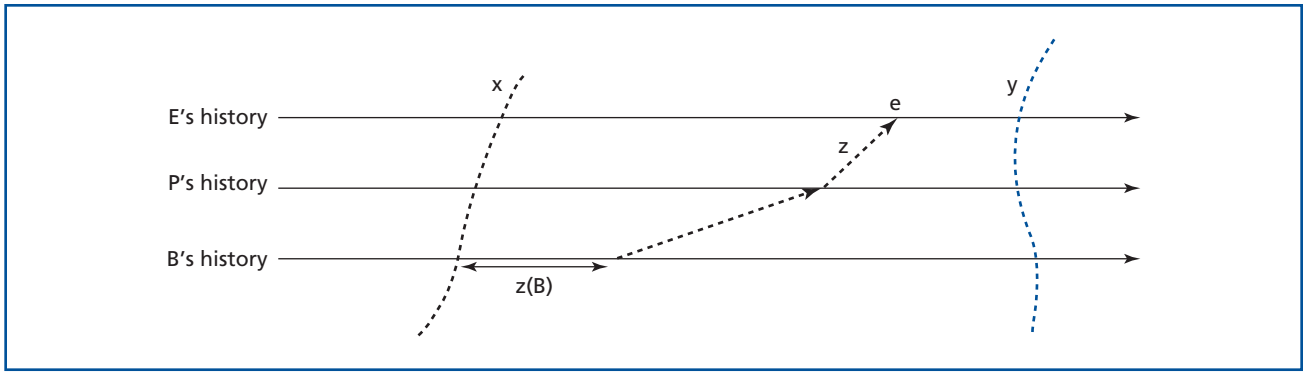


Figure 6.
Illustrating the proof of Theorem 2.

Now z also satisfies the sub-assertion B *knows* (not(E *knows* not(q))). By the downward-closure construction, any event in z(B), the B-sequence of z, is part of a chain <B;P;E>. Consider the downward closure of events in z(B). If there is no chain <P;B> in the interval (x,z), the downward closure will not include any events of P in the interval (x,z), nor will it include any events of E in the interval (x,z)—since, by the structural assumption, an event in E can influence an event in B only via an event in P. Hence, the sequence z′ formed by restricting the sequences of P and E to those in x, and the sequence of B to that in z (i.e., z′ defined by z′(P) = x(P), z′(E) = x(E), and z′(B) = z(B)) is a valid computation (i.e., it is downward-closed). Note that z′ and z are indistinguishable to B. Hence, the assertion B *knows* (not(E *knows* not(q))) also holds for z′, as does its sub-assertion (not(E *knows* not(q))). So E *knows* not(q) fails to hold at z′. But z′ and x are indistinguishable to E since z′(E) = x(E). Hence, the assertion E *knows* not(q) must also *fail* at x. This contradicts the assumption of the claim that E *knows* not(q) holds for x. Thus, there must be a chain <P;B> in (x,z) and therefore a chain <P;B;P;E> in the interval (x,y). **Endproof**

It is possible to construct a specific S and E where a chain of the type in Theorem 2 occurs infinitely often. Let S maintain a counter, initially 0, incrementing at each 'tick' message from E. The only other message from E is a 'query' message from E. For both messages, the response from S is the current value of the counter. Consider an infinite computation, x, in FF where the counter increments infinitely often, and a response is received by E infinitely often. By completeness, this computation must be matched by an infinite failure-free computation, y, of FT. By the bisimularity between FF and FT, which preserves the state of S and the events of E, corresponding points along x and y satisfy the same "E *knows* q" predicates where q is a predicate on S. Let q(k) be the predicate "counter is at least k." In x, for each $k > 0$, there is a point where E *knows* not(q(k)) and a later point where E *knows* q(k). Hence, there are corresponding pairs of points satisfying the same predicates along y. The prefixes induced by each pair of points meet the conditions of Theorem 2, forcing a process chain <P;B;P;E> between those points. Infinitely many of these witnessing chains must be distinct, showing that synchronization is required infinitely often.

Any generic primary-backup protocol must be able to handle this specific service/environment combination. The specific service considered above can be seen as an abstraction of a number of real protocols. For instance, a service which provides stateful load balancing must keep the assignment of sessions to servers up-to-date; this is analogous to the update of the counter. A query is represented by a message in a session which must be directed to the appropriate server. If two states differ, they must differ on the assignment for some session; thus, there exists a message directed to that session for which the response

(the server it is assigned to) is different from the two states.

Related Work

The most closely related work on real time scheduling in primary-backup replication is that by Zou and Jahanian [10]. Their solution is to adjust the admission policy for real-time tasks to account for synchronization delays. This results in fewer tasks being admitted, which effectively reduces throughput. Budhiraja et al. [2] show that there is a non-blocking protocol for their crash + link failure model. This model counts transmission failures against the failure budget—i.e., a protocol with a budget of 1 trivially meets its specification after two transmission failures. We consider transmission failure to be normal, which forces a blocking protocol. This proof is also different from classical impossibility results, such as the impossibility of consensus in a distributed system [5] or the CAP theorem [6], since it does not show that backup in itself is impossible: only that *timely* backup is impossible. A short summary of the proof given here was published in [7]. This paper significantly extends that of [7] by including an extended discussion of the issues and the full proof of a stronger impossibility result.

Conclusions

The impossibility result is an interesting one, as it places a lower bound on a whole class of solutions. However, its real value is that an analysis of the proof can suggest ways in which the negative result may be side stepped. One possibility is a replication mechanism organized differently from the standard primary-backup structure, for instance by allowing backups to communicate directly with the environment. Another is to map the replication mechanism to the application and the distinct characteristics of its network traffic. For instance, an application may communicate with the replication protocol to allow certain messages to be released to the environment without a prior backup synchronization, since the message would not lead to the environment gaining knowledge of a visible predicate. Both possibilities open up new and interesting directions for research.

References

[1] M. C. Browne, E. M. Clarke, and O. Grümberg, "Characterizing Finite Kripke Structures in Propositional Temporal Logic," Theoret. Comput. Sci., 59:1-2 (1988), 115–131.

[2] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg, "The Primary-Backup Approach," Distributed Systems, 2nd ed. (S. Mullender, ed.), ACM Press/Addison-Wesley, New York, 1993, pp. 199–216.

[3] K. M. Chandy and J. Misra, "How Processes Learn," Distrib. Comput., 1:1 (1986), 40–52.

[4] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield, "Remus: High Availability via Asynchronous Virtual Machine Replication," Proc. 5th USENIX Symp. on Networked Syst. Design and Implementation (NSDI '08) (San Francisco, CA, 2008), pp. 161–174.

[5] M. J. Fischer, N. A. Lynch, and M. Paterson, "Impossibility of Distributed Consensus with One Faulty Process," Proc. 2nd ACM SIGACT-SIGMOD Symp. on Principles of Database Syst. (PODS '83) (Atlanta, GA, 1983), pp. 1–7.

[6] S. Gilbert and N. Lynch, "Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services," ACM SIGACT News, 33:2 (2002), 51–59.

[7] P. Koppol, K. S. Namjoshi, T. Stathopoulos, and G. T. Wilfong, "The Inherent Difficulty of Timely Primary-Backup Replication," Proc. 30th ACM SIGACT-SIGOPS Symp. on Principles of Distrib. Comput. (PODC '11) (San Jose, CA, 2011), pp. 349–350.

[8] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," Commun. ACM, 21:7 (1978), 558–565.

[9] F. B. Schneider, "Replication Management Using the State-Machine Approach," Distributed Systems, 2nd ed. (S. Mullender, ed.), ACM Press/Addison-Wesley, New York, 1993, pp. 169–197.

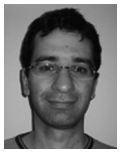
[10] H. Zou and F. Jahanian, "A Real-Time Primary-Backup Replication Service," IEEE Trans. Parallel Distrib. Syst., 10:6 (1999), 533–548.

(Manuscript approved March 2012)

PRAMOD KOPPOL is a technical manager within Alcatel-Lucent's Software, Services & Solutions group in Holmdel, New Jersey. He has a bachelor of engineering degree in computer science and engineering from Osmania University, India; an M.S. in

computer science from Southern Illinois University, Carbondale, Illinois; and a Ph.D. in computer science from North Carolina State University, Raleigh, North Carolina. His primary research focus includes software, protocols, and systems aspects relating to networking and mobile devices, and he has participated in the development, productization and commercialization of several research ideas. He was a founding member and served as head of product development for the OmniAccess® 3500 Nonstop Laptop Guardian, an Alcatel-Lucent Ventures initiative.

KEDAR S. NAMJOSHI is a member of technical staff in the Enabling Computing Technologies research domain at Bell Labs in Murray Hill, New Jersey. He holds Ph.D. and M.S. degrees from the University of Texas at Austin, and a B.Tech degree from the Indian Institute of Technology (IIT), Madras, all in the computing sciences. His research interests span several topics in program analysis and verification, temporal logics, and distributed systems.



THANOS STATHOPOULOS was a member of technical staff in the Enabling Computing Technologies research domain at Bell Labs in Murray Hill, New Jersey when this work was done. He is currently at Google. His general interests lie in software and systems aspects of networks, with emphasis on mobile and embedded devices. Prior to joining Bell Labs, Dr. Stathopoulos was a research scientist in the Electrical Engineering Department at the University of California at Los Angeles (UCLA). He received his M.Sc. in computer science and Ph.D. in computer science from UCLA, working on systems and networking elements of wireless sensor networks, with emphasis on energy-aware design.

GORDON T. WILFONG is a distinguished member of technical staff in the Enabling Computing Technologies research domain at Bell Labs in Murray Hill, New Jersey. His major research interests are in algorithm design and analysis. His current area of focus is networking including protocol analysis, routing, scheduling, and analyzing game theoretic network models. He holds patents in a wide range of areas including protocol design, optical networking, handwriting recognition, security, document analysis, and cross-connect design. He received his B.Sc., First Class Honors, in mathematics from Carleton University, in Ottawa, Ontario, Canada and his M.S. and Ph.D. in computer science from Cornell University in Ithaca, New York. ◆