Availability Study of Dynamic Voting Algorithms

Kyle Ingols* and **Idit Keidar**
MIT Lab for Computer Science,
545 Technology Square, NE43-367, Cambridge, MA 02139, U.S.A.

Abstract

Fault tolerant distributed systems often select a primary component to allow a subset of the processes to function when failures occur. The dynamic voting paradigm defines rules for selecting the primary component adaptively: when a partition occurs, if a majority of the previous primary component is connected, a new and possibly smaller primary is chosen.

Several studies have shown that dynamic voting leads to more available solutions than other paradigms for maintaining a primary component. However, these studies have assumed that every attempt made by the algorithm to form a new primary component terminates successfully. Unfortunately, in real systems, this is not always the case: a change in connectivity can interrupt the algorithm while it is still attempting to form a new primary component; in such cases, algorithms typically block until processes can resolve the outcome of the interrupted attempt.

This paper uses simulations to evaluate the effect of interruptions on the availability of dynamic voting algorithms. We study four dynamic voting algorithms, and identify two important characteristics that impact an algorithm's availability in runs with frequent connectivity changes. First, we show that the number of communication rounds exchanged in an algorithm plays a significant role in the availability achieved, especially in the degradation of availability as connectivity changes become more frequent. Second, we show that the number of processes that need to be present in order to resolve past attempts impacts the availability, especially during long runs with numerous connectivity changes.

*Kyle Ingols' current address is: Oracle Corporation, 200 Fifth Avenue, Waltham, MA 02451.

1 Introduction

Distributed systems typically consist of a group of processes working on a common task. Processes in the group multicast messages to each other. Problems arise when connectivity changes occur, and processes are partitioned into multiple disjoint network components[1]. In many distributed systems, at most one component is permitted to make progress in order to avoid inconsistencies.

Many fault tolerant distributed systems use the *primary component* paradigm to allow a subset of the processes to function when failures and partitions occur. Examples of such systems include group-based toolkits for building distributed applications, such as ISIS [BvR94], Phoenix [MS95], and xAMp [RV92]; algorithms supporting state-machine replication, such as [Lam98, KD96]; and replicated database systems like [EAT89]. Typically, a majority (or quorum) of the processes is chosen to be the primary component. However, in highly dynamic and unreliable networks this is problematic: repeated failures along with processes voluntarily leaving the system may cause majorities to further split up, leaving the system without a primary component. To overcome this problem, the *dynamic voting* paradigm was suggested.

The dynamic voting paradigm defines rules for selecting the primary component adaptively: when a partition occurs, if a majority of the previous primary component is connected, a new and possibly smaller primary is chosen. Thus, each newly formed primary component must contain a majority of the previous one, but not necessarily a majority of the processes.

An important benefit of the dynamic voting paradigm is its flexibility to support a dynamically changing set of processes. With emerging world-wide communication technology, new applications wish to allow users to freely join and leave. Using dynamic voting, such systems can dynamically account for the changes in the set of participants.

The availability of algorithms that use dynamic voting has been extensively studied. Analyses of stochastic models [JM90, CW96], simulations [PL88], and empirical results [AW96] have been used to show that dynamic voting is more available than other paradigms for maintaining a primary component. Specifically, these studies have shown that algorithms that use dynamic voting lead to a primary component being formed more often than algorithms that use a regular quorum-based rule for choosing the primary component.

All of these studies have assumed that every attempt made by an algorithm to form a new primary component terminates successfully. Unfortunately, in a distributed system, this cannot be guaranteed to always be the case: a change in connectivity can interrupt the dynamic voting algorithm while an attempt to form a primary component is in progress. In such cases, dynamic voting algorithms typically *block* until they can *resolve* the outcome of the interrupted attempt. The analyses of the availability of dynamic voting mentioned above did not take the possibility of blocking into consideration, and therefore, the actual system availability is lower than analyzed.

In this paper, we use simulations to study the availability of dynamic voting algorithms, without neglecting the effect of interruptions and blocking. We examine cases in which a sequence of closely clustered changes in connectivity occur in the network, and then the network stabilizes to reach a quiescent state. Connectivity changes can be either network partitions or merging of previously disconnected components. We vary the number and frequency of the connectivity changes. We study how *gracefully* different dynamic voting algorithms degrade when the number and frequency of such changes increase.

The realistic simulation of network connectivity changes is still a subject of much debate and research. The tests were therefore run under a wide variety of conditions, in an effort to cover

[1]A component is sometimes called a partition. In our terminology, a partition splits the network into several components.

most eventualities. However, we did not study cases with only a single network partition. In such a scenario, simply choosing the component with a majority will always succeed. The dynamic voting algorithms come into play in the event of multiple network connectivity changes. Closely clustered connectivity changes mirror the often sporadic nature of network changes. This could simulate situations as simple as a router failing and then returning to service, or any other transient turbulence in the network.

When interrupted, dynamic voting algorithms differ in their resilience: some of the suggested algorithms (e.g., [JM90, Ami95]) may block until all the members of the last attempt to form a primary component become reconnected. Others (e.g., [MS95, YLKD97, DPFLS98, Lam98]) can make progress whenever a majority of the members of the last attempt to form a primary component are present. Algorithms also differ in how long it takes them to resolve the outcome of interrupted attempts to form a primary component, and in their ability or inability to pipeline multiple such attempts.

We study four dynamic voting algorithms: The first algorithm is due to Yeger Lotem et al. [YLKD97]. The second is a variation on the first, due to De Prisco et al. [DPFLS98]. The third is based on the idea of *two phase commit*, similarly to the algorithms suggested in [JM90, Ami95]. The fourth resembles *three phase commit*, similar to ideas presented in [Lam98, MS95]. As a control, we also compare the algorithm with the simple (non-dynamic) majority rule for selecting a primary component.

The set of algorithms we study is representative, but not comprehensive; it is not the goal of this work to study every algorithm ever suggested. Rather, our work illustrates the importance of considering the effect of interruptions when studying the availability of dynamic voting algorithms. Our study points out two parameters that affect availability while there are interruptions. We invite other researchers to use our framework[2] in order to study additional algorithms and to compare them with those studied here.

Our results show that the blocking period has a significant effect on the availability of dynamic voting algorithms in the face of multiple subsequent connectivity changes. We point out two parameters that significantly affect the degradation of availability as the number of connectivity changes rises, and as these changes become more frequent: (1) the number of message rounds conducted by an algorithm; and (2) the number of processes that need be contacted in order to resolve past attempts. We observed poor degradation for algorithms that require many communication rounds, and algorithms that sometimes require a process to hear from all the members of a previous attempt before progress can be made. In contrast, algorithms that use few message rounds and allow progress whenever a majority of the members of the previous attempt reconnects, degrade gracefully as the number of connectivity changes increases, even during lengthy executions with thousands of connectivity changes.

The results emphasize the importance of considering the effect interruptions have on the availability of these algorithms. Previous studies have overlooked the effects of interruptions on the algorithms' availability. We show that interruptions have a tangible effect on the algorithms' availability, and that resilient algorithms with few message rounds will therefore have an edge that has not been previously acknowledged. The insights gained in this work may lead to studies of the availability of other algorithms, for example, atomic commit algorithms, in the face of frequent interruptions.

[2]Our testing framework code is publicly available from http://theory.lcs.mit.edu/~idish/test-env.html.

- The system consists of five processes: a, b, c, d and e. The system partitions into two components: a, b, c and d, e.
- a, b and c attempt to form a new primary component. To this end, they exchange messages.
- a and b form the primary component $\{a, b, c\}$, assuming that process c does so too. However, c detaches before receiving the last message, and therefore is not aware of this primary component. a and b remain connected, while c connects with d and e.
- a and b notice that c detached and form a new primary $\{a, b\}$ (a majority of $\{a, b, c\}$).
- Concurrently, c, d and e form the primary component $\{c, d, e\}$ (a majority of $\{a, b, c, d, e\}$).
- The system now contains two live primary components, which may lead to inconsistencies.

Figure 1: Scenario illustrating inconsistencies in the naive approach.

2 The Studied Algorithms

In this section, we overview the algorithms studied in the paper, and highlight the differences between them. Due to space limitations, we do not include detailed algorithm descriptions here; the interested reader is referred to [YLKD97, DPFLS98, Ing00].

We study four algorithms that use *dynamic linear voting* [JM90] to determine when a set of processes can become the next primary component in the system. Dynamic voting allows a majority of the previous primary component to form a new primary component. Dynamic linear voting also admits a group of processes containing exactly half of the members of the previous primary component if the group contains a designated member of the previous primary (the one with the lowest process-identifier).

In order to form a new primary component, processes need to *agree* to form it. Lacking such agreement, subsequent failures may lead to concurrent existence of two disjoint primary components, as demonstrated by the scenario shown in Figure 1.

In order to avoid such inconsistencies, dynamic voting algorithms have the processes *agree* on the primary component being formed. If connectivity changes occur while the algorithm is trying to reach such agreement, some dynamic voting algorithms (e.g., [JM90, Ami95]) may block until they hear from *all* the members of the last attempt to form a primary component, and do not attempt to form new primary components in the mean time. Others, (e.g., [MS95, YLKD97, DPFLS98]), can make progress whenever a *majority* of the members of the last attempt to form a primary component are present.

In this paper, we study four algorithms based on the dynamic voting principle. In addition, we implemented and tested the simple majority algorithm which declares a primary component whenever a majority of the original processes are present. We now describe the five studied algorithms.

2.1 YKD

The first algorithm we study is due to Yeger Lotem et al. [YLKD97], henceforward YKD. The YKD algorithm overcomes the difficulty demonstrated in the scenario in Figure 1 by keeping track of pending attempts to form new primaries. In the example above, the YKD algorithm guarantees that if a and b succeed in forming $\{a, b, c\}$, then c is aware of this possibility. From c's point of view, the primary component $\{a, b, c\}$ is *ambiguous*: it might have or might have not been formed by a and b. While there are pending attempts, the YKD algorithm may initiate further attempts to form primary components. Thus, there can be multiple pending attempts that a process attempted to form but detached before actually forming them. Every process records, along with the last primary component it successfully formed, later primary components that it attempted to form. These ambiguous attempts are taken into account in later attempts to form a primary component. Once a primary component is successfully formed, all ambiguous attempts are deleted.

In addition, the YKD algorithm employs an optimization that reduces the number of ambiguous attempts that processes store and send to each other. The optimization reduces the worst-case number of attempts from exponential in the number of processes to linear. In practice, however, the number of attempts retained is very small: In experiments presented in [Ing00] we observe that very few ambiguous attempts are actually retained. Even in highly unstable runs, with up to 64 processes participating, the number of ambiguous attempts retained by the YKD algorithm was dominantly zero. In fact, the highest observed number in over 600,000 64-process runs was four, and it occurred only twice. The optimization does not affect the availability of the algorithm, only the amount of storage utilized and the size of exchanged messages.

The YKD algorithm works as follows: Whenever a connectivity change occurs, the algorithm is invoked to try to make the new connected component the primary one. To this end, the processes conduct two message rounds. In the first round, the processes exchange state – sending each other their ambiguous attempts, last primary components, and so on. Based on this state, each process checks if the new component can become a primary one, by checking if it contains a majority of the members of the last formed primary component, and also of all ambiguous attempts retained after it. If the component passes these checks, the processes then *attempt* to make it a primary; they record it as a pending attempt and exchange a second round of messages. If this second round is successfully received by all processes, then the primary component is completed. If the second round is not received (due to another connectivity change), then the attempted primary remains *ambiguous*.

2.2 DFLS: unoptimized YKD with an extra round

The second studied algorithm is a variation on YKD due to De Prisco et al. [DPFLS98], henceforward called DFLS. The DFLS algorithm was introduced in order to simplify the correctness proof of YKD. It is a variation on the YKD algorithm that does not implement the optimization, and also does not delete ambiguous attempts immediately when a new primary is formed. Instead, it waits for another message exchange round to complete in the new formed primary before deleting them. This delay in deleting ambiguous attempts may limit the system availability, since these attempts act as constraints that limit future primary component choices.

2.3 1-pending: one ambiguous attempt only

We also study an algorithm which does not attempt to form a new primary component while there is a pending attempt. We call this algorithm 1-pending. Whenever there is a pending ambiguous attempt, 1-pending tries to *resolve* the pending ambiguous attempt before attempting to form a new primary. 1-pending resolves a pending attempt by learning the outcome of that attempt from other processes. In the worst case, a process needs to hear from all the members of the pending attempt in order to resolve its outcome. If it cannot resolve the attempt, 1-pending blocks. In comparison, YKD is sometimes able to make process even if it cannot resolve the previous ambiguous attempt at the time. 1-pending is very similar to *two phase commit* based algorithms such as those suggested in [JM90, Ami95].

2.4 MR1p: majority-resilient 1-pending

As mentioned above, 1-pending may need to hear from every process in an ambiguous attempt before the attempt can be resolved. Dynamic voting algorithms that employ *three phase commit* like mechanisms (for example, those suggested in [Lam98, MS95]), are always able to resolve an ambiguous attempt when hearing from a majority of the attempt's members. We have implemented such an algorithm; we refer to this algorithm as *Majority-Resilient 1-pending*, or *MR1p*. Like 1-pending, it can retain at most one ambiguous attempt. However, it is able to resolve its ambiguous attempt in more cases than 1-pending can.

When there are no pending ambiguous attempts, MR1p forms a primary component using two message rounds, similar to those of YKD. When there is a pending ambiguous attempt, MR1p first runs three message rounds to resolve the status of the pending attempt. Once the attempt is resolved, two additional rounds are then run to attempt to form a new primary. The algorithm used by MR1p to form and resolve primary component is very similar to the Consensus algorithm of [Lam98], and to *three phase commit* [KD98]. For more details on our implementation of MR1p, please see [Ing00].

2.5 Simple majority

Additionally, as a control, we tested a simple majority-based primary component algorithm which does not involve message exchange. This algorithm declares a primary whenever a majority of the processes are present. It also declares a primary for a group containing half the processes if that group contains a designated member (the one with the lowest process-identifier). In this respect, the algorithm can be seen as a static version of linear voting. This majority rule is the most available static quorum system [PW95].

This simple algorithm requires almost no state other than process-identifiers, sends no messages, and is very fast. The dynamic voting principle and algorithms based on it were created in an effort to improve upon this simple idea in highly failure-prone networks.

2.6 Comparison of the dynamic voting algorithms

The studied algorithms differ in two important ways: First, they differ in the number of message rounds they execute. Second, they differ in their resilience, that is, in the number of members of a failed attempt that need be contacted in order to resolve the attempt.

The number of message rounds each algorithm requires to run is of critical importance. Algorithms which require many message rounds are more likely to be interrupted by further connectivity changes. YKD and 1-pending require only two message rounds. DFLS requires three rounds — two to form a primary component, and a third before it deletes ambiguous attempts. MR1p requires only two rounds when no pending view is present, but requires five rounds if a pending view must also be resolved.

While there are no pending ambiguous attempts, all the dynamic voting algorithms attempt to form a new primary if a majority of the members of the previously formed primary component are present. When there are ambiguous attempts, YKD, DFLS, and MR1p can make progress whenever a majority of the members of all ambiguous attempts are present (with MR1p there is at most one such attempt). 1-pending is the least resilient of the studied algorithms; it requires hearing from all the members of a pending attempt in order to make progress.

3 The Testing Framework

The dynamic voting algorithms are implemented as C++ classes with a set of call-back routines and no inherent communication abilities. Programs using a dynamic voting algorithm are expected to call the algorithm's call-back routines with every message received, every message about to be sent, and every connectivity change. The call-back routines return with messages that need to be sent to other processes, to be in turn handled by the appropriate call-back handlers at the recipients. The call-back routines declare a primary component when the algorithm successfully terminates.

Because the algorithms are individual classes with no dependencies on any given communication system, the testing system easily simulates an arbitrary number of processes by creating multiple instances of the algorithm. The testing environment consists of a driver loop implemented in C++. The driver loop routes all messages among the multiple instances of the algorithm without using the network or any communication system. It does this by polling individual processes for messages to send, and then immediately delivering those messages to the other processes. The driver loop also supports fault injection and statistics gathering during the simulation.

The user specifies two simulation parameters: the number of connectivity changes to inject in each run, and the frequency of these changes. The frequency of changes is specified as the mean number of message rounds which are successfully executed between two subsequent connectivity changes. The mean is obtained using an appropriate probability p, so that a connectivity change is injected at each step with probability p.

The testing system begins each simulation with all the processes mutually connected. The processes are then allowed to exchange messages. The driver loop chooses whether to inject a connectivity change at each step, according to the failure probability. Once the desired number of changes have been introduced, the driver loop allows the processes to exchange messages without further interruptions until the system reaches a stable state. The driver loop then prints out final statistics, the most relevant of which is the presence or absence of a primary component.

A connectivity change is either a network partition, where processes in one network component are divided into two smaller components, or a merge, where two components are unified to produce one. The driver loop has an equal likelihood of generating either of these changes[3]. The components to be partitioned or merged are chosen at random. Partitions do not necessarily happen evenly – the percentage of processes which are moved to the new component is determined at random each time.

4 Primary Component Availability Measurements

We compare the availability of five algorithms: YKD, DFLS, 1-pending, MR1p, and simple majority, as explained in Section 2. We also ran the tests for an unoptimized version of YKD, that is, YKD without the optimization that reduces the number of ambiguous attempts retained. The availability of the unoptimized YKD was identical to that of YKD, (with the optimization), as expected. Therefore, we do not plot the availability of the unoptimized YKD separately.

We chose to simulate 64 processes. We also ran the same tests with 32 and 48 processes to see if the availability is affected by scaling the number of processes. The results obtained with 32 and 48 processes were almost identical to those obtained with 64. Therefore, we do not present them here.

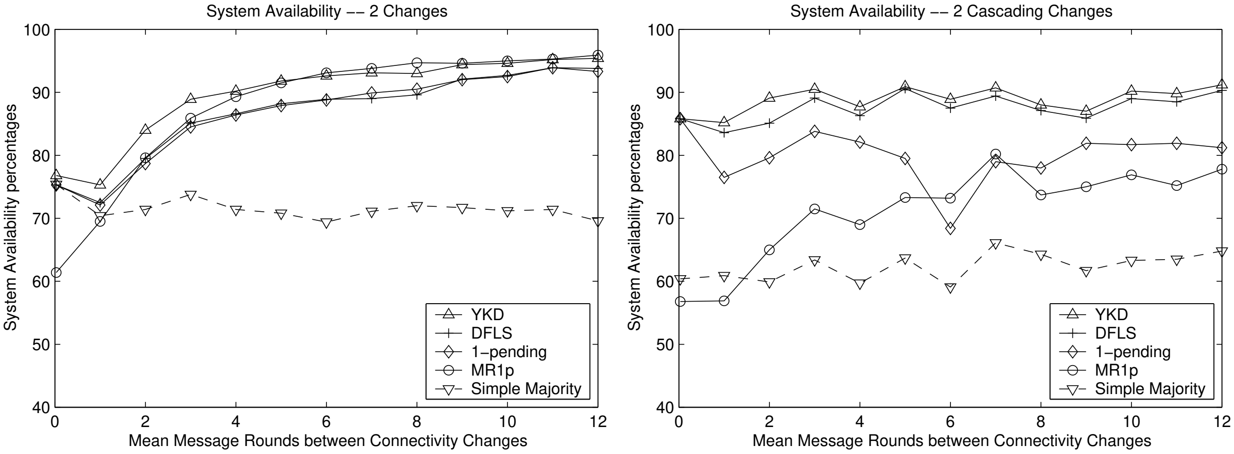
[3]Given that such a change is possible, of course – one cannot perform a merge unless there are at least two components present, and one cannot perform a partition unless there is a component with at least two processes.

We simulated three different numbers of network connectivity changes per run: two, six, and twelve. For each of these, we ran each of the algorithms with connectivity change rates varying from nearly zero to twelve mean message rounds between changes.

Each case (specified by the algorithm, the number of connectivity changes and the rate), was simulated in 1000 runs. The runs were different due to the use of randomization. The same random sequence was used to test each of the algorithms. The results for each case were then summarized as a percentage, showing how many of the runs resulted in the successful formation of a primary component at the end of the run.

We ran two types of tests: "fresh start" tests, where each run begins from the same initial state, and "cascading" tests, where each run starts in the state at which the previous run ends. The fresh start tests capture the effect of interruptions on a single invocation of an algorithm. The cascading tests capture the build-up that can occur during a lengthy execution, where earlier invocations can leave the algorithm in a blocking state. The cascading tests reflect the algorithms' expected behavior in realistic long-term executions. Therefore, algorithms that exhibit lower availability in these tests will be less available in practice.

The results for all five algorithms, for two, six, and twelve connectivity changes are presented in Figures 2, 3, and 4, respectively. The x-axis represents the average number of communication rounds that occur between a pair of injected connectivity changes, derived from the appropriate failure probability. The y-axis depicts the percentage of the 1000 simulated runs for this failure probability that resulted in successful formation of a primary component.



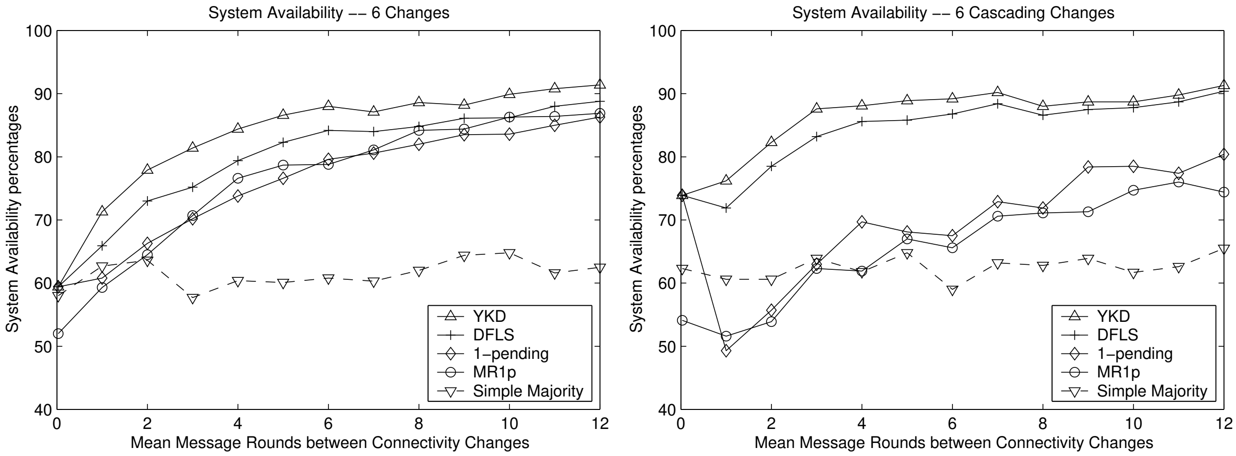
(a) "Fresh Start". (b) "Cascading".

Figure 2: System availability with 2 connectivity changes.

On the extreme left side of the graphs, the connectivity changes are so tightly spaced the algorithms are often unable to exchange any additional information. On the extreme right side of the graphs, the connectivity changes are so widely spaced that the algorithms are rarely interrupted. As expected, the availability improves as the conditions become more stable.

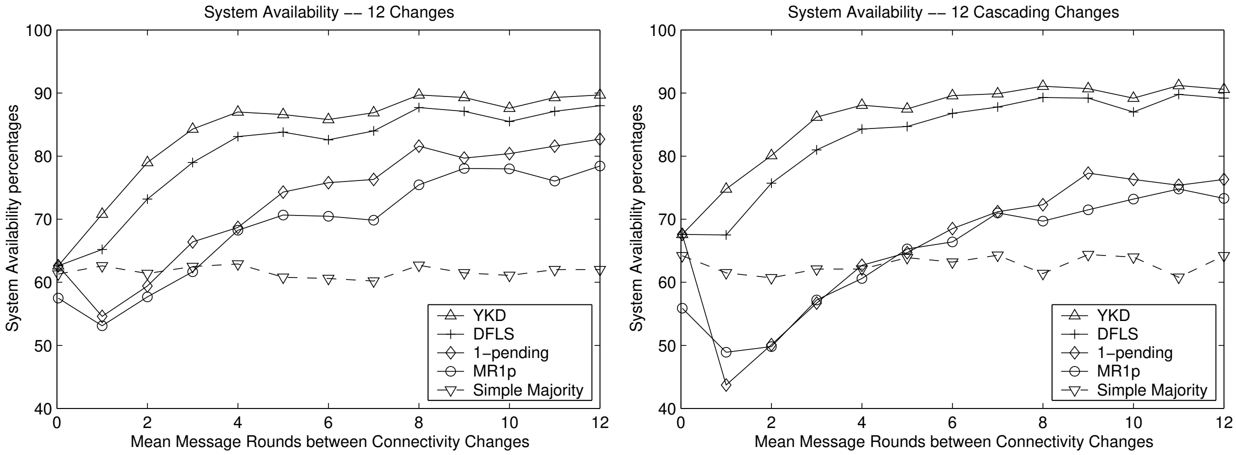
In all cases, the algorithms are shown to be about as available as the simple majority algorithm when the connectivity changes occur rapidly. This is simply due to the fact that rapid changes do not allow the algorithms any time to exchange information between connectivity changes, and they have no additional knowledge with which to decide on a primary component.

For a moderate to high mean time between changes, YKD and DFLS are most available, with YKD being slightly more available than DFLS; in approximately 3% of the runs, YKD succeeds



(a) “Fresh Start”. (b) “Cascading”.

Figure 3: System availability with 6 connectivity changes.



(a) “Fresh Start”. (b) “Cascading”.

Figure 4: System availability with 12 connectivity changes.

in forming a primary whereas DFLS does not. This difference stems from the additional round of messages required by DFLS before an attempt can be deleted. As long as the attempt is not deleted, it imposes extra constraints which limit the system's choice of future primary components. Both algorithms – YKD and DFLS – degrade gracefully as the number of connectivity changes increases, that is, their availability is almost unaffected.

The 1-pending and MR1p algorithms are significantly less available than YKD and DFLS. Furthermore, their availability degrades drastically as the number of connectivity changes increases. This degradation is due to the fact that these algorithms cannot make any progress whenever they cannot resolve an ambiguous attempt. In the worst case, 1-pending requires hearing the outcome of its ambiguous attempt from all of its members. Thus, permanent absence of some member of the latest ambiguous attempt may cause eternal blocking. Although MR1p requires only a majority, it requires five message rounds to complete, making it more prone to interruption. This emphasizes the value of YKD's ability to make progress even when some of the algorithm's prior ambiguous attempts cannot be resolved.

In the "fresh start" tests with two connectivity changes, we observe that MR1p is almost as available as YKD. This is due to the fact that there can be at most one ambiguous attempt to resolve between the two connectivity changes, and that YKD and MR1p are equally powerful at resolving a single ambiguous attempt. However, as the connectivity changes increase in number and frequency, MR1p is less available than all other algorithms studied. Although it is able to resolve ambiguous attempts more often than 1-pending, it requires a very large number of message rounds to execute. The algorithm is therefore interrupted so frequently compared to the others that it is unable to readily make progress.

YKD and DFLS provide almost identical availability in tests with cascading failures as in tests with a fresh start. These results indicate that even if the algorithms are run for extensive periods of time, their availability does not degrade. Note that for the two, six and twelve connectivity change cases, the results are computed over a running period with 2,000, 6,000, and 12,000 connectivity changes, respectively.

In contrast, the availability of the 1-pending algorithm shows major degradation in the cascading situation. In cases with numerous frequent connectivity changes, the algorithm is often even less available than the simple majority. This shows that if the 1-pending algorithm is run for extensive periods of time, its availability continues to decrease. This makes the algorithm inappropriate for use in systems with lengthy life periods.

The MR1p algorithm has further difficulties when the failures are allowed to cascade. Although it is able to resolve its single ambiguous attempt more quickly than 1-pending can, it is still hampered by the large number of message rounds it requires in order to form a primary. In addition, YKD is sometimes able to make progress even when one or more ambiguous attempts are present. MR1p does not have this luxury.

5 Conclusions

We have compared the availability of four dynamic voting algorithms in the face of frequent connectivity changes. Our measurements show that interruptions have a significant effect on the availability of dynamic voting algorithms in the face of multiple subsequent connectivity changes. This effect was overlooked by previous availability analyses of such algorithms (e.g., [JM90, PL88, CW96]).

We have shown that the number of processes that need be contacted in order to resolve past ambiguous attempts significantly affects the availability, especially in long executions with numerous connectivity changes. A two phase commit like algorithm, 1-pending, experiences major degradation as the number and frequency of connectivity changes increase. In highly unstable runs with cascading connectivity changes, 1-pending is even less available than the simple majority algorithm. This is due to the fact that 1-pending sometimes requires a process to hear from all the members of the previous primary component before progress can be made.

We have also observed that the number of message rounds executed by an algorithm has a major effect on availability, especially the degradation of availability as there are more connectivity changes, and as these changes become more frequent. A three phase commit like algorithm, MR1p, was shown to degrade drastically as the number and frequency of connectivity changes increase. This degradation is because the MR1p algorithm is highly vulnerable to interruptions, due to the large number of message rounds it executes.

In contrast, an algorithm that uses few communication rounds and also makes progress whenever a majority of the members of pending attempts are present degrades gracefully as the number and frequency of connectivity changes increase. The YKD [YLKD97] and DFLS [DPFLS98] algorithms are nearly as available in runs with cascading connectivity changes as they are in runs with a fresh start. This feature makes the algorithms highly appropriate for deployment in real systems with extensive life spans.

We hope that the insights gained in this work will lead to similar studies of the availability of other algorithms in the face of frequent interruptions. For example, it may be interesting to study the availability of different atomic commit algorithms when there are multiple connectivity changes.

Acknowledgments

We thank Alan Fekete and Alex Shvartsman for many helpful suggestions.

References

[Ami95] Y. Amir. *Replication Using Group Communication Over a Partitioned Network.* PhD thesis, Institute of Computer Science, Hebrew University, Jerusalem, Israel, 1995.

[AW96] Y. Amir and A. Wool. Evaluating quorum systems over the internet. In *IEEE Fault-Tolerant Computing Symposium (FTCS)*, pages 26–35, June 1996.

[BvR94] K. Birman and R. van Renesse. *Reliable Distributed Computing with the Isis Toolkit.* IEEE Computer Society Press, 1994.

[CW96] I. R. Chen and C. Wang, D. Analyzing dynamic voting using petri nets. In *IEEE International Symposium on Reliable Distributed Systems (SRDS)*, 1996.

[DPFLS98] R. De Prisco, A. Fekete, N. Lynch, and A. Shvartsman. A dynamic view-oriented group communication service. In *17th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 227–236, June 1998.

[EAT89] A. El Abbadi and S. Toueg. Maintaining availability in partitioned replicated databases. *ACM Transactions on Database Systems*, 14(2):264–290, June 1989.

[Ing00] Kyle W. Ingols. Availability Study of Dynamic Voting Algorithms. Master's thesis, MIT, May 2000. Master of Engineering. Available at:
http://theory.lcs.mit.edu/ idish/Abstracts/ingols-thesis.html.

[JM90] S. Jajodia and D. Mutchler. Dynamic voting algorithms for maintaining the consistency of a replicated database. *ACM Transactions on Database Systems*, 15(2):230–280, 1990.

[KD96] I. Keidar and D. Dolev. Efficient message ordering in dynamic networks. In *15th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 68–76, May 1996.

[KD98] I. Keidar and D. Dolev. Increasing the resilience of distributed and replicated database systems. *Journal of Computer and System Sciences special issue with selected papers from ACM SIGACT-SIGMOD Symposium on Principles of Database Systems (PODS) 1995*, 57(3):309–324, December 1998.

[Lam98] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998. Also Research Report 49, Digital Equipment Corporation Systems Research Center, Palo Alto, CA, September 1989.

[MS95] C. Malloth and A. Schiper. View synchronous communication in large scale networks. In *2nd Open Workshop of the ESPRIT project BROADCAST (Number 6360)*, July 1995.

[PL88] J.F. Paris and D.D.E. Long. Efficient dynamic voting algorithms. In *13th International Conference on Very Large Data Bases (VLDB)*, pages 268–275, 1988.

[PW95] D. Peleg and A. Wool. Availability of quorum systems. *Inform. Comput.*, 123(2):210–223, 1995.

[RV92] L. Rodrigues and P. Verissimo. xAMp, a protocol suite for group communication. RT /43-92, INESC, January 1992.

[YLKD97] E. Yeger Lotem, I. Keidar, and D. Dolev. Dynamic voting for consistent primary components. In *16th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 63–71, August 1997.