Availability Study of Dynamic Voting Algorithms

Kyle Ingols               Idit Keidar

MIT Lab for Computer Science.
545 Technology Square, NE43-367, Cambridge, MA 02139, U.S.A.
E-mail: kwi@mit.edu, idish@theory.lcs.mit.edu, phone: (617) 253 9302.

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 is more available than any other paradigm
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. Unfortu-
nately, in real systems, this is not always the case: if a change in connectivity occurs while the
dynamic voting algorithm is still running, algorithms typically block until processes can resolve
the outcome of the interrupted attempt.

This paper measures the effect of blocking on the availability of dynamic voting algorithms,
using simulations. The paper shows that the number of processes required in order to resolve
past attempts and the speed at which past attempts are resolved play a role in the availability
achieved; they especially affect the degradation of availability as there are more connectivity
changes, and as these changes become more frequent.
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 [3], Phoenix [7], and xAMP [9], and replicated database systems like [5]. 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.

Stochastic models analysis [6], simulations [8], and empirical results [2] have been used to show that dynamic voting is more available than any other paradigm for maintaining a 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 real systems, this is not always the case: if a change in connectivity occurs while an attempt to form a primary component is in progress, algorithms typically block until they can resolve the outcome of the interrupted attempt.

\(^1\)A component is sometimes called a partition. In our terminology, a partition splits the network into several components.
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 measure the effect of blocking on the availability of dynamic voting algorithms. 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 most eventualities. However, we did not study cases with only a single network failure. 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 multiple pieces of the network segmenting almost simultaneously, or any other transient turbulence in the network.

Dynamic voting algorithms differ in the length of their blocking period: some of the suggested algorithms (e.g., [6, 1]) may block until all the members of the last primary become reconnected; others (e.g., [7, 10, 4]) can make progress whenever a majority of the last primary becomes reconnected. Algorithms also differ in how long it takes them to resolve the outcome of interrupted attempts, and in their ability or inability to pipeline multiple attempts.

We focus on the dynamic voting algorithm of Yeger Lotem et al. [10]. We compare its availability with that of two variations on it – one variation due to De Prisco et al. [4], and a second variation which is similar (although not identical) to the dynamic voting algorithms suggested in [6, 1]. 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. We cannot study every algorithm ever suggested, nor can we be sure to implement every algorithm in a manner faithful with the authors’ intent. We invite other researchers to use our framework\textsuperscript{2} in

\textsuperscript{2}Our testing framework code is publicly available from http://theory.lcs.mit.edu/~idish/test-env.html.
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. The number of processes that need be contacted in order to resolve past attempts significantly affects the degradation of availability as the number of connectivity changes rises, and as these changes become more frequent. Algorithms that sometimes require a process to hear from all the members of a previous attempt before progress can be made degrade drastically as the number of connectivity changes increases. Furthermore, in lengthy executions with numerous connectivity changes, the availability of these algorithms constantly degrades. In contrast, algorithms which allow progress whenever a majority of the members of the previous attempt reconnects degrade gracefully as the number of connectivity changes increases, and do not degrade during lengthy executions.

2 Background: The Studied Algorithms

In this section, we overview the dynamic voting algorithms studied in the paper, and explain the differences between them. Due to space limitations, we do not include detailed algorithm descriptions here; the interested reader is referred to [10, 4].

We study several algorithms that use *dynamic linear voting* [6] 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 process (the one with the lowest process-id).

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 following typical scenario:

- 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.

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., [6, 1]) may block until they hear from all the members of the last primary component, and do not attempt to form new primary components in the mean time.

The algorithm of principal study is the dynamic voting algorithm of [10], henceforward, the YKD algorithm. This algorithm overcomes the difficulty demonstrated in the scenario above 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. Unlike previously suggested dynamic voting algorithms, the YKD algorithm does initiate new attempts to form primary components while there are pending attempts. Every process records, along with the last primary component it formed, later primary components that it attempted to form but detached before actually forming them. 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 [10] 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 our experiments we observe that very few ambiguous attempts are actually retained. Even in highly unstable runs, with 48 processes participating, the number of ambiguous attempts retained by the YKD algorithm was dominantly zero, and never exceeded four (cf. Section 5). The optimization does not affect the availability of
the algorithm, only the amount of storage utilized and the size of exchanged messages.

The algorithm of [4], henceforward DFLS, is a variation on the YKD algorithm which 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 occur in the newly formed primary before deleting them. This delay in deleting ambiguous attempts limits the system availability, since these attempts act as constraints that limit future primary component choices. In our experiments, we observed that in approximately 5% of the runs, the YKD algorithm succeeds in forming a primary component when the DFLS algorithm does not (cf. Section 4). Both algorithms degrade gracefully as the number and frequency of connectivity changes increase. Furthermore, we showed that the YKD and DFLS algorithms can run for extensive periods of time, experiencing thousands of connectivity changes, and still show no degradation in availability.

We also study a variant of the YKD algorithm which does not attempt to form a new primary component while there is a pending attempt. We call this algorithm 1-pending. 1-pending blocks whenever there is a pending ambiguous attempt; it tries to resolve the pending ambiguous attempt before attempting to form a new primary. A pending attempt can be resolved by a process 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. 1-pending is very similar to the algorithms suggested in [6, 1]. Our experiments show that the 1-pending algorithm is significantly less available than the YKD and DFLS algorithms. We also showed that if 1-pending is run for extensive periods of time, its availability constantly degrades (cf. Section 4).

Additionally, as a control, we tested the 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. As with dynamic linear voting, a set containing exactly half of the processes can be a primary component if it contains a designated process.

3 Testing System Implementation

The dynamic voting algorithms are implemented as C++ classes with no inherent communication abilities. Programs using the algorithm are expected to call the algorithm with every message received, every message about to be sent, and every connectivity change.
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 be executed between two subsequent connectivity changes. The mean is obtained using an appropriate uniform 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 while the driver loop injects connectivity changes with the appropriate 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\(^3\). Partitions do not necessarily happen evenly – the percentage of processes which are moved to the new component is determined at random each time.

Due to the CPU-intensive nature of these tests, the system ran on multiple machines and submitted results over the Internet to a central machine for collection and analysis. After receipt, the data passed through a series of Perl scripts for tabulation and summarizing. Matlab was then used to perform the final plots and simple manipulation of the data.

\(^3\)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.
4 Primary Component Availability Measurements

We compared the availability of four algorithms: YKD, DFLS, 1-pending, and simple majority. 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 48 processes. We also ran the same tests with 32 processes to see if the availability is affected by scaling the number of processes. The results obtained with 32 processes were almost identical to those obtained with 48. We do not include them here for lack of space.

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 500 or 1000 runs (depending on the test type, see below). 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, with 500 runs each starting from the same initial state, and “cascading” tests, with 1000 runs each starting from the state at which the previous run ends. The results are presented in Figures 1(a) and 1(b) respectively.

With both tests, 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 is more available than DFLS; in
approximately 5% of the runs, YKD succeeds 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 degrade gracefully as the number of connectivity changes increase, that is, their availability is almost unaffected.

The 1-pending algorithm is significantly less available than the other tested dynamic voting algorithms. Furthermore, its availability degrades drastically as the number of connectivity changes increases. This degradation is due to the fact that 1-pending cannot make any progress whenever
it cannot resolve an ambiguous attempt, which in the worst case, requires hearing the outcome of that attempt from every member of that attempt. Thus, permanent absence of some member of the latest attempt may cause eternal blocking. This emphasizes the value of YKD’s ability to make progress even when some of the algorithm’s prior attempts cannot be resolved.

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, these 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 dramatically degrades 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 constantly decreases. This makes the algorithm inappropriate for use in systems with lengthy life periods.

5 The Number of Pending Ambiguous Attempts

The number of ambiguous attempts retained by an algorithm affects not only the memory consumption but also the size of messages being exchanged, as the algorithms exchange information about ambiguous attempts. The message size affects system performance in a way that was not accounted for in the availability tests above.

In this section we study the number of ambiguous attempts retained by three algorithms: YKD, the unoptimized version of YKD, and DFLS. We do not study the number of ambiguous attempts retained by 1-pending as it is at most one.

The statistics were collected by one of the processes during the fresh start tests described above. For each run, the process reported the number of ambiguous attempts stored when the network situation stabilized at the end of the run. The results were then summarized for each 500-run case, (a case is specified by the algorithm, the number of connectivity changes and the rate).

In Figure 2, we show the percentage of runs for which the algorithm retained ambiguous attempts for each case. Each data-point is comprised of three bars. In order from left to right,
Figure 2: Ambiguous attempts with YKD, Unoptimized YKD, and DFLS with 2, 6, and 12 connectivity changes.

The bars represent YKD, unoptimized YKD, and DFLS. The total height of the bar indicates the percentage of the 500 runs in which ambiguous attempts were retained. The bar is further divided into blocks, which indicate the actual number of ambiguous attempts retained. The bottom block represents a single retained attempt, the second — two and so forth.

The most striking phenomenon observed is how few ambiguous attempts are retained. The theoretical worst-case number of attempts that could be retained by DFLS and the unoptimized YKD is exponential in the number of processes, and for YKD it is linear. However, in all of our runs, including the highly unstable cascading ones, the number of ambiguous attempts retained
never exceeded 8 for DFLS, and never exceeded 4 for YKD. The number of retained attempts was dominantly zero. This demonstrates how unlikely the worst-case scenarios truly are.

Please note that at the conclusion of a successful run, none of the algorithms retains any ambiguous attempts at all. Therefore, the bars are higher for DFLS simply due to the fact that succeeds in less runs, that is, it is less available. The bars for YKD and unoptimized YKD are identical in height since these algorithms have identical availability. However, the unoptimized YKD retains a higher number of ambiguous attempts, on average.

6 Conclusions

We have compared the availability of three dynamic voting algorithms. Our measurements show that the blocking period has 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., [6, 8]).

We have shown that the number of processes that need be contacted in order to resolve past ambiguous attempts significantly affects the availability, especially the degradation of availability as there are more connectivity changes, and as these changes become more frequent. The 1-pending algorithm degrades drastically as the number and frequency of connectivity changes increase. In highly unstable runs with cascading connectivity changes, it 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.

In contrast, the YKD algorithm [10] degrades gracefully as the number and frequency of connectivity changes increase. It is nearly as available in runs with cascading connectivity changes as it is in runs with a fresh start. This feature makes the algorithm highly appropriate for deployment in real systems with extensive life spans.

The DFLS algorithm [4] degrades as gracefully as the YKD algorithm. However, it is less available than YKD with all failure patterns. This illustrates the effect of the speed at which attempts are resolved on the availability.
Acknowledgments

We thank Alex Shvartsman for many helpful suggestions.

References


