Availability Study of Dynamic Voting Algorithms

by

Kyle W. Ingols

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Masters of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2000

Author . . . .
Department of Electrical Engineering and
Computer Science
May 5, 2000

Certified by . . . .
Idit Keidar
Postdoctoral Associate
Thesis Supervisor

Accepted by . . . .
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

Availability Study of Dynamic Voting Algorithms

by

Kyle W. Ingols

Submitted to the Department of Electrical Engineering and
Computer Science
on May 5, 2000, in partial fulfillment of the
requirements for the degree of
Masters of Engineering in Electrical Engineering and
Computer Science

Abstract

Fault tolerant distributed systems often select a primary component to allow a subset of the processes to function when failures occur. Several studies have examined algorithms for selecting primary components. 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: if a change in connectivity occurs while the algorithm is still running, algorithms typically *block* until processes can *resolve* the outcome of the interrupted attempt.

This thesis first presents a framework for the implementation of primary component algorithms. This framework is used to implement several algorithms based on the dynamic voting principle. The thesis then shows, using simulations, that an algorithm's performance is highly affected by interruptions; availability degrades as more connectivity changes occur, and as these changes become more frequent.

Thesis Supervisor: Idit Keidar
Title: Postdoctoral Associate

Acknowledgments

There were so many people inconvenienced by the ever-present “thesisitis” hanging over my head that I couldn’t hope to name all of them, even if I did expound upon each individual’s virtues in a vain effort to expand this document’s length. That said, I shall try for an abridged version.

First and foremost, I must thank my parents, Bill and Debra Ingols, for getting me here. How they were able to put up with me for this long I will never know. I owe them everything, and will try in vain to repay them.

Thanks to Idit Keidar, my tireless thesis advisor, for providing the gentle (and sometimes less than subtle) kicks in the behind needed to keep me on track and working. She is the most stringent, thorough, and proactive proofreader I have yet known. Hopefully my prose will someday approach hers in clarity, though I suspect that my writing abilities will “drastically degrade” as I age. . . ;)

Thanks to Miss Christine for being patient with me when I spent every waking hour working on this thing, and more than a few non-waking hours slumped over it!

Last but not least, thanks to Alex Shvartsman for many helpful suggestions in the writing of the paper from which this thesis is developed.

That’s all, folks. I’m going to bed now. . .

Contents

List of Figures

Chapter 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 [4], Phoenix [9], and xAMp [11], and replicated database systems like [6]. 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

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

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 [7], simulations [10], and empirical results [3] 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. 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 order to examine the algorithms' performance under stress, we first design a simple, general interface which can be used to communicate with the algorithm. This API is designed to be free of unnecessary dependencies on specific communication services, allowing the user to choose. We then use this interface to implement five algorithms for selecting a primary component. Once the algorithms are implemented and tested, we run simulations on them and examine the results.

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.

When interrupted, dynamic voting algorithms differ in the length of their blocking period: some of the suggested algorithms (e.g., [7, 1]) may block until all the members of the last primary become reconnected; others (e.g., [9, 12, 5, 8]) 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 to form a primary component, and in their ability or inability to pipeline multiple such attempts.

We focus on the dynamic voting algorithm of Yeger Lotem et al. [12], hereafter called YKD. We compare its availability with that of four variations on it – one which removes some memory-saving optimizations, a second variation due to De Prisco et al. [5], a third variation which is similar (although not identical) to the dynamic voting algorithms suggested in [7, 1], and a fourth which is based on ideas presented in [8, 9]. As a control, we also compare the algorithm with the simple (non-dynamic) majority rule for selecting a primary component.

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 degrades even further. 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 with thousands of connectivity changes.

The results emphasize the importance of considering the effect interruptions have on the performance of these algorithms. Previous studies have overlooked the effects of interruptions on the algorithms' availability, concentrating instead on examining ideal conditions. We show that interruptions have a tangible effect on the algorithms' availability, and that algorithms with few message rounds will therefore have an edge that has not been previously acknowledged.

Other algorithms designed to choose primary components may also demonstrate robust behavior when exposed to connectivity changes. We naturally cannot implement and consider all of them here, nor can we hope to explore every interesting failure scenario. We therefore present our testing framework and algorithm implementations[2] for the use of other researchers.

In Chapter 2, we discuss the implementation of the algorithms and their testing system in more detail. The workings of the algorithms themselves are covered in Chapter 3. We then examine the specific tests run and the results from them in Chapter 4. Chapter 5 concludes the thesis with a summary of our results, and thoughts about possible future work.

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

Chapter 2

Algorithm Implementation

The initial thrust of our work was the implementation of the algorithm of Yeger Lotem et al. [12] for real-world use. Henceforth we shall refer to the algorithm by the abbreviation YKD. The intention was to integrate the YKD algorithm into a complete system upon which an application developer could base a fault-tolerant distributed application. To this end, YKD was initially paired with Transis [2], a group communication service which provides notification of connectivity changes.

Once the algorithm was completed, we designed a testing system to help prove that the algorithm was implemented correctly. In addition, we expanded the testing system to collect detailed statistics for a variety of scenarios in order to analyze the actual performance of the algorithm.

2.1 The Algorithm-to-Application Interface

The interface required by YKD is very simple. The algorithm requires only the ability to broadcast messages, receive messages and reports of connectivity changes, and maintain state. Any interface which provides those services will enable YKD to function properly. The choice of Transis is arbitrary, but not restrictive; any group communication service which has reliable multicast and can report connectivity changes will work.

Typically, group communication services such as [2, 4, 9, 11] report connectivity changes as *views*. A view is nothing more than a list of all of the processes which are currently connected. The only artifact of the choice of Transis is that the interface uses the Transis `view` structure to represent the list of processes in a view. This decision does not limit us to using only Transis; the Transis view structure is simple and easily portable. It should therefore be simple to seamlessly translate other group communication services' view structures to the Transis structure.

The implemented dynamic voting algorithms are *event-driven*, i.e., the only time the algorithm's state changes is when it sends or receives a message or view. This means that there is no need to continually poll the algorithm, checking to see if it wishes to send a message. If the algorithm has no need to send a message now, the only reason it would need to send one in the future is the arrival of further information (a message from somewhere else or a connectivity change).

The interface is designed to "piggyback" information onto messages sent by the application – the application is required to pass all messages it wishes to send through the algorithm before transmission. When the algorithm has received new information (either a message or a new view), the application is expected to offer to send a message.

Thus, we can model the interface to YKD, or any general algorithm designed to choose primary components, easily as a C++ class. Pseudocode for this class is suggested in Figure 2-1.

The algorithm must be started with a list of all of the processes in the very first view – in other words, the initial view in which the processes begin together. The algorithm expects that every view after the first will contain only processes which were present in the first view[1]. Once the system is running, the application is expected to pass each incoming message through the `incomingMessage` method before looking at it and to pass each outgoing message through the `outgoingMessagePoll` method before transmission. This allows the algorithm to "piggyback" information

[1] The YKD algorithm is actually capable of handling new processes which join the system after the initial view is established, but that ability is not studied or used here.

```
class PrimaryComponentAlgorithm
{
  constructor( transisView initialView );
   // specific algorithms may need other initial information which
   // could be provided to the constructor here
  destructor();

  Message *incomingMessage( Message m, int senderID );
      // returns the same message, with the algorithm's
      // information stripped from it

  Message *outgoingMessagePoll( Message *tobesent );
      // returns NULL if no modification to the message is made,
      // else returns the new message to send instead

  viewChanged( transisView newView );

  int inPrimary();
```

Figure 2-1: Pseudocode of algorithm interface.

onto messages sent by the application, and to remove that information from received messages before they are passed on to the application. The application never sees the extra information exchanged by the algorithm.

In addition, each time a message is received, the application should immediately query the `outgoingMessagePoll` function, even if the application itself has nothing it wants to send. This gives the algorithm an opportunity to communicate even if the application using it is idle.

The application can then use the `inPrimary` call at its leisure to determine whether or not it is in a primary component. As with the other methods, there is never a need to poll `inPrimary`; this state can only change if and when new information arrives (a message or connectivity change). Therefore, the application need only check `inPrimary` after a new message has arrived.

By implementing the algorithm as an independent entity with no inherent communication abilities of its own, we free the algorithm from dependence on any one

```
void applicationSendMessage( Message *m)
{
   Message *m2 = myDynVotingObject->outgoingMessagePoll( m);
   if( m2 != NULL)
     BroadcastMessageOnNetwork( m2);
   else
     BroadcastMessageOnNetwork( m);
}


Message *applicationReceiveMessage( Message *m)
{
   Message *m2 = myDynVotingObject->incomingMessage( m);
   Message *mEmpty = new Empty Message;
   Message *m3 = myDynVotingObject->outgoingMessagePoll( mEmpty);
   if( m3 != NULL)
     BroadcastMessageOnNetwork( m3);

   return m2;
}
```

Figure 2-2: Pseudocode of application using interface.

communication service. Instead, the burden is placed on the application developer to integrate the two. This integration is extremely simple, and is demonstrated in Figure 2-2.

2.2 The Testing and Simulation System

Implementing the algorithm this way also makes testing simple. The testing system easily simulates an arbitrary number of processes by creating multiple instances of the algorithm. It requires no actual networking abilities at all – the system takes advantage of the fact that the algorithm does not possess any inherent communication ability.

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.

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[2]. Partitions do not necessarily happen evenly – the percentage of processes which are moved to the new component is determined at random each time.

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.

These tests also served as a very extensive trial-by-fire of the algorithm's implementation. Each of the algorithms was subjected to over 1,310,000 connectivity changes, and none of them demonstrated an inconsistency, leaked memory, or crashed.

[2] 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.

Every process in a view agreed on whether or not that view was a primary, and at all times there was at most one primary component declared.

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.

Chapter 3

Algorithm Descriptions

We study several algorithms that use *dynamic linear voting* [7] 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 scenario shown in Figure 3-1.

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., [7, 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.

We study five algorithms based on the dynamic voting principle. The first, YKD, is the main algorithm of study. We also explore four variations of the YKD algorithm, three of which are similar to other algorithms suggested in the literature.

In addition, we implemented and tested the simple majority algorithm in order to provide a baseline from which the performance of the other algorithms can be

- 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 3-1: Scenario illustrating inconsistencies in the naive approach.

measured. This algorithm declares a primary component whenever a majority of the original processes are present.

3.1 YKD

The algorithm of principal study is the dynamic voting algorithm of [12]. This algorithm overcomes the difficulty demonstrated in the scenario in Figure 3-1 by keeping track of pending ambiguous sessions 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 order to operate successfully, a process p using YKD must maintain a fairly extensive amount of local state. The state makes frequent use of a construct that we will call a *session*. A session is nothing more than a view with a number attached to it, corresponding to an session to form a primary component. These numbers are used by YKD to determine the order in which views occurred.

The state which a process p running YKD retains is as follows:

- The *initial view* of the process – all of the processes present when the algorithm began. This initial view is the same for all participating processes. We will denote it as `W`.
- The *last primary component* the process successfully formed. Denoted simply as `lastPrimary`.
- The last primary component the process formed *with a given process*. This information is kept as a collection of sessions. We denote `lastFormed(q)` to indicate the last primary component p formed which included q. Initially, all of these entries equal `W`.
- The process's *ambiguous sessions*. This is a list of all of the process's ambiguous sessions. This will be denoted (obviously enough) as `AmbiguousSessions`.
- A *session number*, initially zero, which is used to number new sessions. This is denoted as `sessionNumber`.
- A simple boolean flag stating whether or not I am presently in a primary component. This is denoted `inPrimary`.

Whenever a connectivity change occurs, the processes in the new view participate in two message rounds. In the first round, the processes exchange all of their internal state – sending each other their ambiguous sessions, last primary components, and so on. If the processes decide to attempt to make the new view a primary component, a second round of messages is sent. 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 becomes *ambiguous*.

The algorithm is able to perform its work in only two message rounds because each process receives the information of all of the other processes. Therefore, each process in the view is working with the same knowledge in a deterministic fashion. A process can be confident that if it decides to attempt a primary, all other processes in the view will make the same decision.

The pseudocode for YKD is presented in Figure 3-2. In this and all other pseudocodes, we denote the process doing these tasks as process p. The main block of pseudocode uses four primitive operations, shown in Figure 3-3 on page 21 and Figure 3-4 on page 22.

The resolution rules embodied by the procedures in Figure 3-3 on page 21 allow the process to update its internal state with respect to the activities of other processes. For example, if process p suffers a connectivity change and is isolated, it can use these resolution rules to update its internal state when it is able to merge back with other processes again. Process p is thus able to learn about the sessions which occurred in its absence.

Once YKD is finished reconciling its state with the state information of all of the other processes, the algorithm must then decide whether or not to attempt a new primary component with the current view. It does this by `COMPUTE`ing additional information from all of the received info. With this combined information, the process can `DECIDE` conclusively whether or not it is safe to declare the current view a primary component. These procedures are shown in Figure 3-4 on page 22.

The decision relies heavily on the dynamic voting principle; it only proceeds if the new session is a `SUBQUORUM` of the previous primary component and of all ambiguous sessions – that is, the new session has a majority of the processes which were in the previous primary component, and which were in every other potential primary component. The `SUBQUORUM` procedure is defined in Figure 3-4 on page 22.

If the previous primary splits precisely in half, then the side which contains the "lexically smallest" process of the previous primary may remain the primary. "Lexi-

```
if a new view V is received,

 isPrimary = FALSE
 send state (sessionNumber, ambiguousSessions, lastPrimary,
             and lastFormed) to everyone in V

 once everyone else's state arrives,
   LEARN about the ambiguousSessions, if possible
   RESOLVE the up-to-date information from other processes, if possible
   COMPUTE maxSession, maxPrimary, and maxAmbiguousSessions

   if p DECIDEs to form a primary,
     sessionNumber = maxSession + 1
     ambiguousSessions = ambiguousSessions +
                           new session( V and sessionNumber)
     send attempt message to everyone in V

   if p gets attempt messages from everyone in V,
     lastPrimary = new session( V and sessionNumber)
     ambiguousSessions = NONE
     isPrimary = TRUE
     for every q in V,
       lastFormed(q) = new session( V and sessionNumber)
```

Figure 3-2: Pseudocode for the main process of YKD.

cally smallest" can be defined in any convenient way; one potential method is to sort based on numeric IP address and process ID. This provides an unambiguous way to decide between the two equal-sized views.

```
to LEARN about the ambiguousSessions:

  for a session S,
    if q is in V, and
       S.Number < V.Number, and
       q is in S, and
       p is in S, and  // we are in S ourselves!
       p attempted to form S,
    then
       if lastFormed(q).Number = S.Number,
       then
         p learns that q formed S
       if lastFormed(q).Number < S.Number,
       then
         p learns that q did not form S


to RESOLVE the up-to-date information from other processes:

  ACCEPT rule:
    for a given session S,
      if p is in S, and
         S.Number > lastPrimary.Number, and
         S was formed by one of its members (some other process
            lists S as its lastPrimary or one of its lastFormed)
      then
         lastPrimary = S
         for every q in S,
            lastFormed(q) = S

  DELETE rule:
    for a given ambiguous session S,
      if no member of p formed S, or
         there exists some other session F, such that
              p is in F, and
              F.Number > S.Number, and
              F was formed by one of its members,
      then
         remove S from the ambiguousSessions
```

Figure 3-3: The LEARN and RESOLVE procedures within YKD.

```
to COMPUTE maxSession, maxPrimary, and maxAmbiguousSessions:

  maxSession = the largest sessionNumber of any process

  maxPrimary = the lastPrimary which has the highest Number.

  maxAmbiguousSessions =
       all of the combined ambiguous sessions from the all processes,
       given that the session's Number is greater than
       maxPrimary's Number.


to DECIDE whether or not the current view V can be a primary:

  if V is a SUBQUORUM of maxPrimary, and
     V is also a SUBQUORUM of every session in maxAmbiguousSessions,
  then
     YOU MAY FORM A PRIMARY
  else
     YOU MAY NOT FORM A PRIMARY


to determine if X is a SUBQUORUM of Y:

  if more than half the processes in Y are also in X,
  then TRUE.
  if exactly half of the processes in Y are also in X, and
    the lexically smallest process is also in X,
  then
    TRUE.
  else
    FALSE.
```

Figure 3-4: The `COMPUTE`, `DECIDE`, and `SUBQUORUM` procedures within YKD.

3.2 Variants of YKD

In addition to studying YKD, several variants of the algorithm were also studied in order to examine the importance of various pieces of the YKD algorithm. The first, *unoptimized YKD*, explores the benefits of YKD's complex learning system. The algorithm is no more or less available than YKD, but it does not try to resolve ambiguous sessions while running. The second, *DFLS*, keeps ambiguous sessions for an additional round, allowing us to measure the relative importance of keeping the number of required message rounds minimal. The third and fourth, *1-pending* and *MR1p*, examine the effects of requiring the algorithm to retain at most one ambiguous session at a time. The algorithms differ in their worst-case resolution: 1-pending may require all of the processes from the ambiguous session to reconnect before the session can be resolved, while MR1p requires only a majority of them.

3.2.1 Unoptimized YKD

The YKD algorithm employs an optimization (a part of `RESOLVE` and `LEARN` from Figure 3-3 on page 21) that reduces the number of ambiguous sessions processes store and send to each other. The optimization reduces the worst-case number of ambiguous sessions retained from exponential in the number of processes to linear. This optimization does not provide additional information – it merely helps remove redundant information. Therefore optimization does not affect the availability of the algorithm, only the amount of storage utilized and the size of exchanged messages.

In practice, however, the number of sessions retained is very small. In our experiments we observe that very few ambiguous sessions are actually retained. Even in highly unstable runs, with 64 processes participating, the number of ambiguous sessions retained by the YKD algorithm was dominantly zero, and never exceeded four. The unoptimized YKD also dominantly retained zero, and never exceeded nine (cf. Section 4.2).

3.2.2 DFLS: Unoptimized YKD with an Extra Round

The algorithm of [5], henceforward DFLS, is a variation on the YKD algorithm which does not implement the optimization, and also does not delete ambiguous sessions immediately when a new primary is formed. Instead, it waits for another message exchange round to occur in the new formed primary before deleting them. This delay in deleting ambiguous sessions limits the system availability, since these sessions act as constraints that limit future primary component choices. In our experiments, we observed that in approximately 3% of the runs, the YKD algorithm succeeds in forming a primary component when the DFLS algorithm does not (cf. Section 4.1). Both algorithms degrade gracefully as the number and frequency of connectivity changes increase. Furthermore, we show 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.

3.2.3 1-pending: YKD with One Ambiguous Session

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. In this respect, 1-pending blocks whenever there is a pending ambiguous session; it tries to *resolve* the pending ambiguous session before attempting to form a new primary. YKD, on the other hand, is sometimes able to make process even if it cannot resolve the previous ambiguous session at the time. A pending session can be resolved by a process by learning the outcome of that session from other processes. In the worst case, a process needs to hear from all the members of the pending session in order to resolve its outcome. 1-pending is very similar to the algorithms suggested in [7, 1]. Our experiments show that the 1-pending algorithm is significantly less available than the YKD and DFLS algorithms. We also show that if 1-pending is run for extensive periods of time, its availability further degrades (cf. Section 4.1).

3.2.4 MR1p: Majority-Resilient 1-pending

In the worst case, 1-pending needs to hear from every process in the ¡ambiguous session before the session can be resolved. A dynamic voting algorithm extremely similar to the algorithms presented in [8, 9] is able to resolve such an ambiguous session with only a majority of the ambiguous session's members in the worst case. We refer to this algorithm as Majority-Resilient 1-pending, or MR1p. It, like 1-pending, can retain at most one ambiguous session. However, it is able to resolve its ambiguous session more quickly than 1-pending can.

A process p running the MR1p algorithm retains the following state:

- The *primary component* the process p most recently formed. This will be denoted as `cur_primary`.
- The process's *ambiguous session*. MR1p retains at most one ambiguous session. This is a view that it attempted to declare as a primary, but was unable to form before the algorithm was interrupted. This will be denoted as `ambiguousSession`.
- A *number*, initially zero, which is used to number certain status messages. This is denoted as `num`.
- A simple boolean flag stating whether or not p is presently in a primary component. This is denoted `inPrimary`.
- A status flag indicating which stage of the algorithm p is presently in. This is used to inform other processes how far p progressed in its efforts to form a primary component from its pending view. This flag is denoted simply as `status`.
- Every *formed primary component* the algorithm has ever successfully made. This can be optimized to contain only a reasonable, bounded number of these components. We denote this list of components as `formedViews`.

Running MR1p requires five message rounds when a pending ambiguous session is present, and two rounds (numbers 4 and 5) when no pending ambiguous session must be resolved. They are as follows:

1. Send to everyone your single `ambiguousSession`.

2. Send to everyone what you know about everyone else's `ambiguousSession`.

3. Send to everyone your call on how your `ambiguousSession` should be resolved. Resolve it if a majority agrees with you.

4. If the new view is a `SUBQUORUM` of the `cur_primary`, send to everyone a request to declare the current view to be a primary component.

5. If all processes have sent a message in step 4, send everyone an attempt message. Declare the new view to be a primary component when a majority of the processes in it have sent a message in step 5.

When the algorithm begins running, it has experienced only the initial view (all processes together). Therefore, `formedViews` contains only the initial view, `cur_primary` is set to the initial view, and `isPrimary` is true. The numeric status variable `num` is initially set to zero. The `status` flag is set to `none`.

The pseudocode for the algorithm is presented below. Notice that the MR1p algorithm utilizes the `SUBQUORUM` primitive which was previously defined in the YKD code in Figure 3-4 on page 22.

```
Upon view V
   is_primary = false
   if ambiguousSession
      send <ambiguousSession, num, status> to all
   else (ambiguousSession = null)
      try_new

Upon receipt of <V, 1> from all members of V
   status = attempt; num = 2
   Send <attempt, V> to all

Upon receipt of <attempt, V> from majority of V
   cur_primary = V; is_primary = true
   ambiguousSession = null; num = 0; status = none
   add V to formedViews

Upon receipt of <V, n, s> from some process
   if V is the same as ambiguousSession
      send <V, status>
        // status is either 'sent,' 'attempt,' or 'try_fail'
   if V is in formedViews, and p is in V
```

```
      send <V, formed>
   if V is not in formedViews, p is in V
      send <V, aborted>
   otherwise
      do nothing

Upon receipt of <V, formed> from some process
   cur_primary = V; is_primary = true
   add V to formedViews
   try_new

Upon receipt of <V, aborted> from some process
   try_new

Upon receipt of <ambiguousSession, ?, ?>
                              from majority of ambiguousSession
   let num be (the maximum number in all such messages) + 1
   let status be the status associated with one of the maxima
   if status = sent then status = try_fail
   send <status, V>

Upon receipt of <try_fail, V> from majority of V
   try_new

Subroutine try_new
   if SUBQUORUM(cur_primary, V)
      send <V, 1> to all
      ambiguousSession = V; num = 1; status = sent
   else
      ambiguousSession = null; num = 0; status = none
```

The basic algorithm does not include any optimizations for the removal of entries from `formedViews`. This unfortunately leaves the size of this collection of views unbounded, making the simple implementation highly unsuited to continuous usage. One simple optimization can be made, however: whenever a new primary is formed which is the same as the original view, all other `formedViews` can be discarded. Further optimizations could be made to more tightly bind the number of `formedViews` which the algorithm can be forced to retain, but this simple optimization is sufficient to make the long-term simulation of the algorithm feasible.

3.3 Simple Majority: a Simple, Stateless Algorithm

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. If the processes happen to split into two groups of identical size, the lexical ordering technique used by YKD (see the end of Section 3.1) may also be used to decide between them.

This simple algorithm requires almost no state other than that required to do the lexical ordering, sends no messages, and is very fast. The dynamic voting principle and the algorithms based on it were created in an effort to improve upon this simple idea. We therefore present the simple majority as a baseline from which we can compare the performance of the other studied algorithms.

3.4 Comparison of Algorithms

The YKD algorithm is a fairly complex one, requiring the transfer and management of the list of ambiguous sessions. An ambiguous session is roughly $2n$ bits in length, where n is the number of processes in the system. Theoretically, each process can keep up to $O(n)$ ambiguous sessions in the worst case. In practice, however, the number retained is dominantly zero or one. In fact, the highest observed number in over 600,000 64-process runs was four, and it occurred only twice (cf. Section 4.2). During the information exchange, each process receives all the information from every other process and must iterate through all of it. The total amount of information which must be transmitted does not exceed two kilobytes during these 64-process trials.

Unoptimized YKD and DFLS, which lack the optimizations from YKD, tend to retain more ambiguous sessions, and therefore take longer to run. The maximum observed number of ambiguous sessions retained was nine, and that occurred only nine times in over 600,000 runs of the algorithm. The optimization pays off for YKD because it has fewer ambiguous sessions which it must transmit across the network and analyze upon receipt. The relationship between the number of ambiguous sessions retained by the three algorithms is explored in Section 4.2.

The number of message rounds each algorithm requires to run is also of critical importance. Algorithms which require many message rounds are more likely to be interrupted by further connectivity changes. YKD, unoptimized YKD, and 1-pending require only two message rounds. DFLS requires three, due to the extra round before ambiguous sessions are removed. MR1p requires only two rounds when no pending view is present, but requires five rounds if a pending view must also be resolved.

The availability of the algorithms differs as well. The algorithms which do not pipeline attempts to form a primary, 1-pending and MR1p, are less available than those capable of handling multiple ambiguous sessions at once, such as DFLS and YKD. DFLS and YKD improve upon MR1p's performance by considering ways to proceed even if an ambiguous session cannot be resolved. The results of the availability studies are presented in the next section.

Chapter 4

Measurements and Results

4.1 Primary Component Availability Measurements

We compare the availability of five algorithms: YKD, DFLS, 1-pending, MR1p, and simple majority. We also ran the tests for unoptimized YKD, that is, YKD without the optimization that reduces the number of ambiguous sessions retained. The availability of unoptimized YKD was identical to that of YKD, 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.

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" results are presented in Figures 4-1, 4-2, and 4-3. The "cascading" results are presented in Figures 4-4, 4-5, and 4-6.

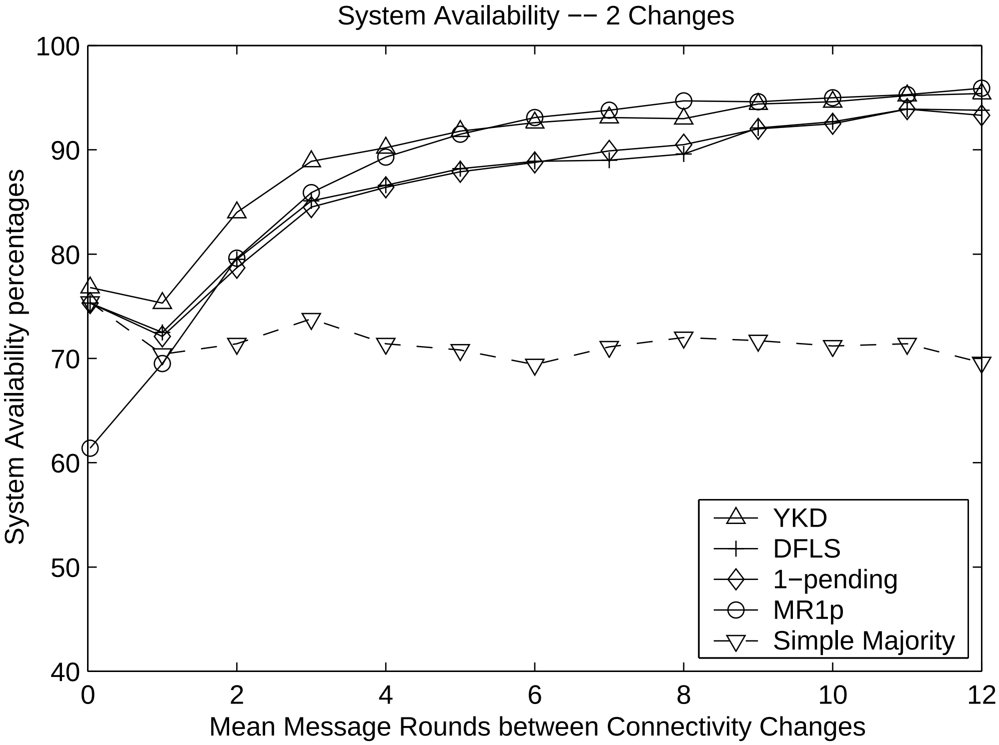

Figure 4-1: System availability with 2 connectivity changes.
"Fresh Start" – each run begins brand-new in the original state.

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

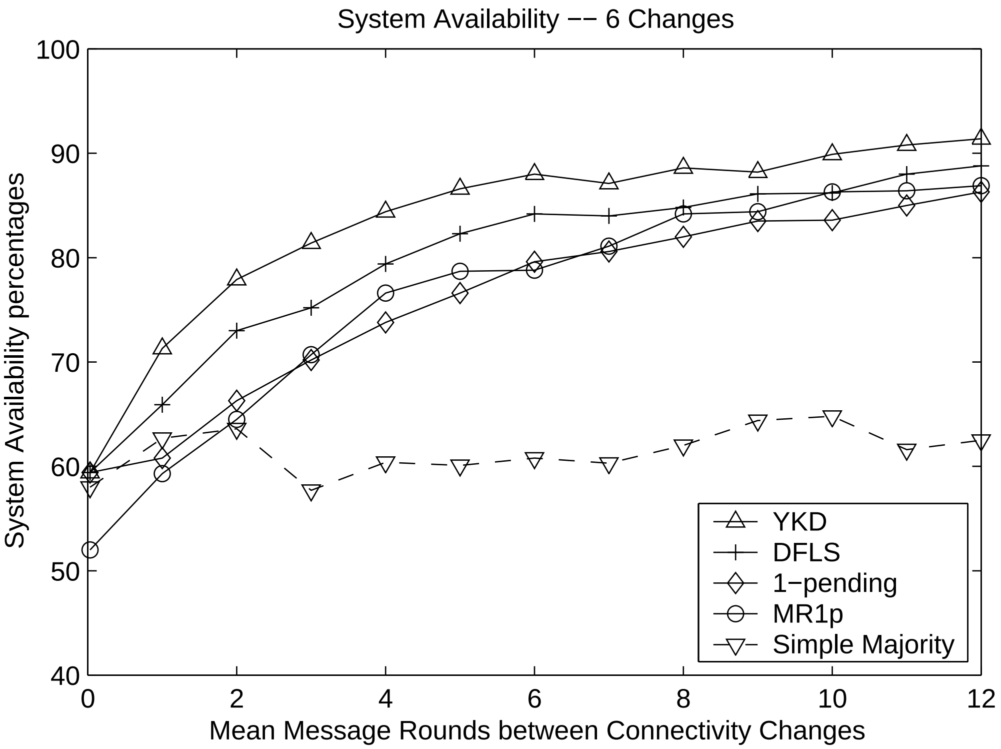

Figure 4-2: System availability with 6 connectivity changes.
"Fresh Start" – each run begins brand-new in the original state.

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 3% 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 ambiguous session can be deleted. As long as the ambiguous session 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 increases, that is, their availability is almost unaffected. These results illustrate the importance of minimizing the number of required message rounds. By running quickly, an algorithm is less likely to be interrupted during its execution.

The 1-pending and MR1p algorithms are significantly less available than YKD and DFLS. Furthermore, their availability degrades drastically as the number of con-

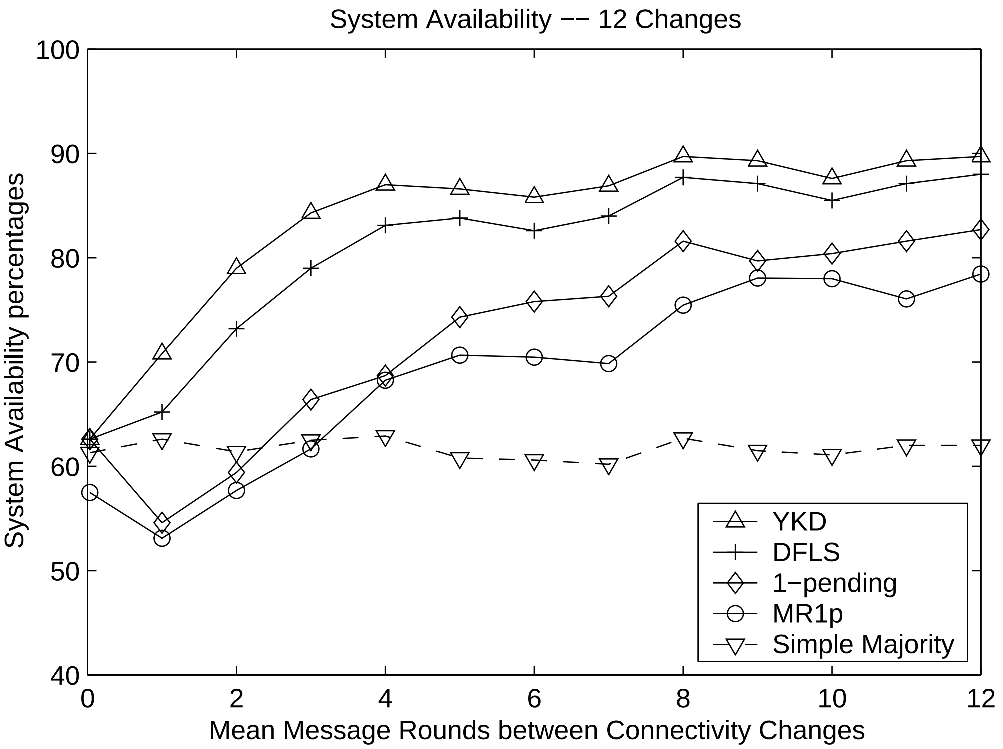

Figure 4-3: System availability with 12 connectivity changes.
"Fresh Start" – each run begins brand-new in the original state.

nectivity changes increases. This degradation is due to the fact that these algorithms cannot make any progress whenever they cannot resolve an ambiguous session. In the worst case, 1-pending requires hearing the outcome of its ambiguous session from all of its members. Thus, permanent absence of some member of the latest ambiguous session 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 sessions cannot be resolved.

In the "fresh start" tests with two connectivity changes, we observe that MR1p is almost as available as to YKD. This is due to the fact that there can be at most one ambiguous session to resolve between the two connectivity changes, and that YKD and MR1p are equally powerful at resolving a single ambiguous session.

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

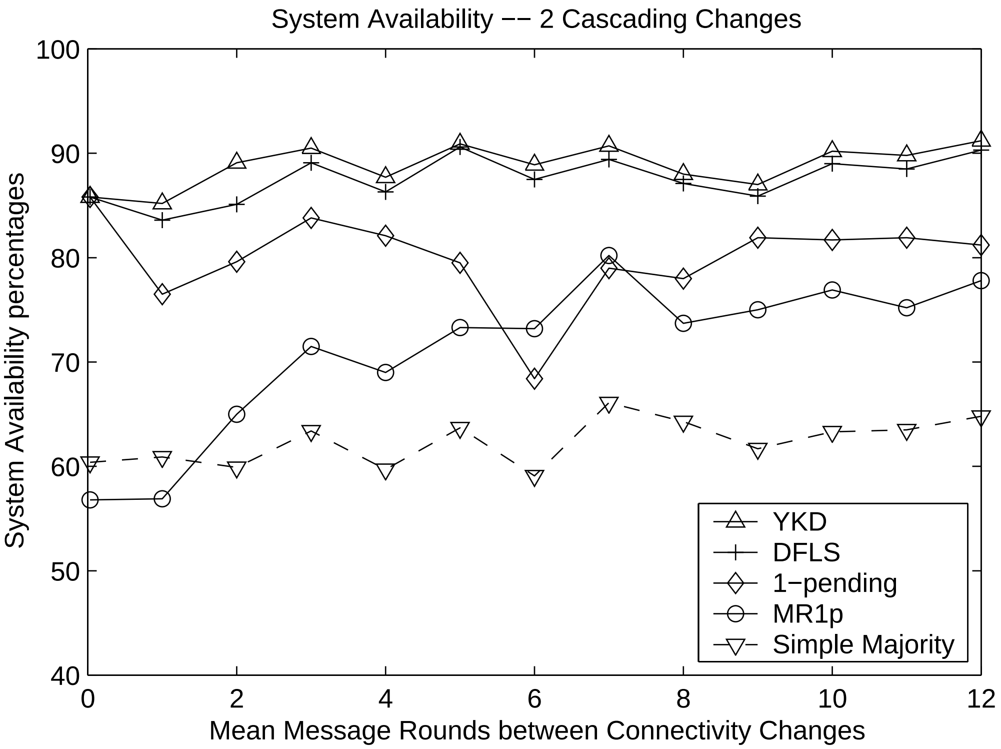

Figure 4-4: System availability with 2 cascading connectivity changes. "Cascading" – each run begins where the previous ends.

ambiguous sessions more often than 1-pending, it requires a very large number of message rounds to execute. The algorithm is 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, 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 continues to decrease. This makes the algorithm inappropriate for use in systems with lengthy life periods.

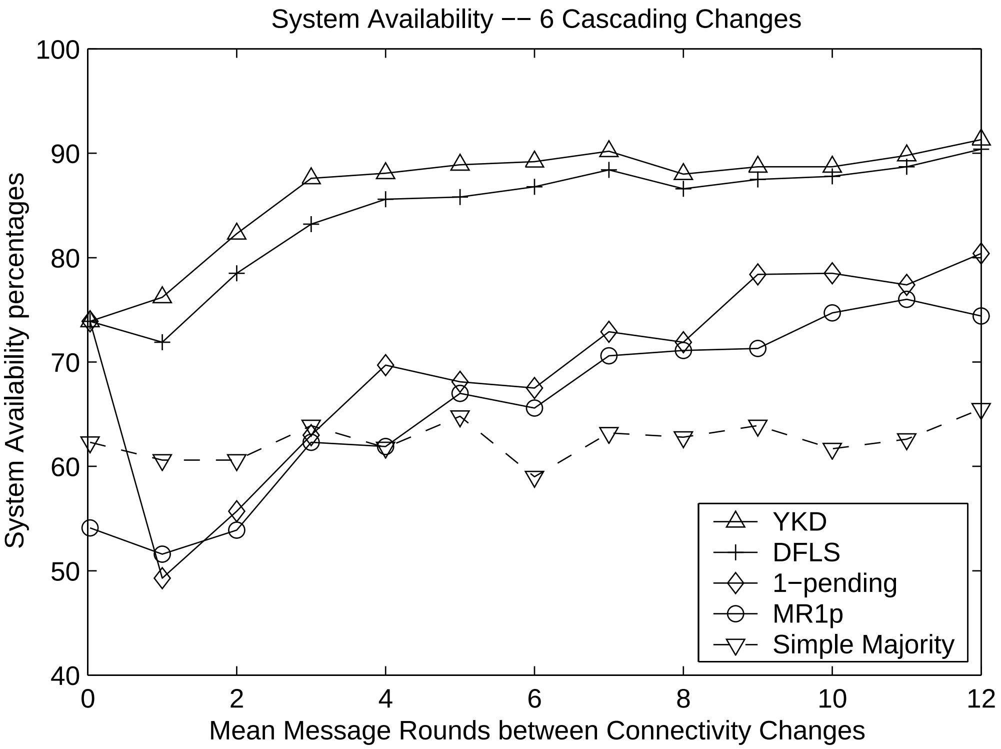

Figure 4-5: System availability with 6 cascading connectivity changes. "Cascading" – each run begins where the previous ends.

The MR1p algorithm has further difficulties when the failures are allowed to cascade. Although it is able to resolve its single ambiguous session 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 sessions are present. MR1p does not have this luxury.

4.2 Measurements of Pending Ambiguous Sessions

The number of ambiguous sessions 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 sessions. The message size affects system performance in a way that was not accounted for in the availability tests above.

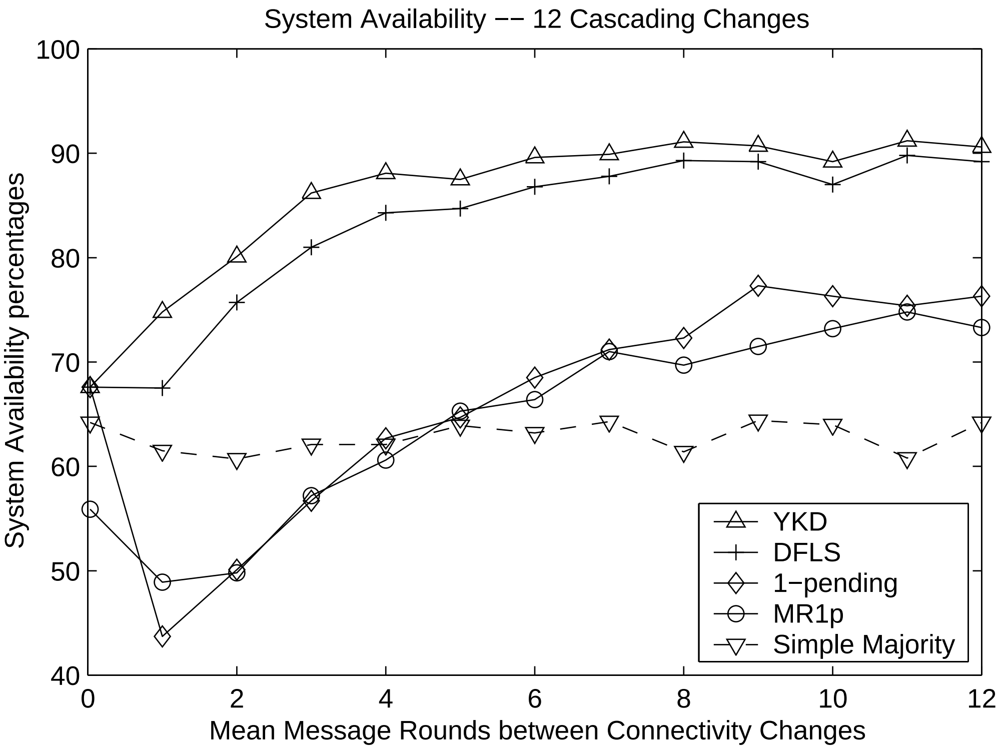

Figure 4-6: System availability with 12 cascading connectivity changes. "Cascading" – each run begins where the previous ends.

In this section we study the number of ambiguous sessions retained by three algorithms: YKD, the unoptimized version of YKD, and DFLS. We do not study the number of ambiguous sessions retained by 1-pending or MR1p as it is at most one.

The statistics were collected by one of the processes during the fresh start tests described above. The cascading tests exhibit similar behavior. For each run, the process reported both the number of ambiguous sessions stored when the network situation stabilized at the end of the run and the number of ambiguous sessions present each time a connectivity change occurred. The results were then summarized for each 1000-run case, (a case is specified by the algorithm, the number of connectivity changes and the rate).

In Figure 4-7, we show the percentage of runs for which the algorithm retained ambiguous sessions for each case. Figure 4-8 shows the percentage of connectivity changes at which the algorithm retained ambiguous sessions for each case. Each data-point is comprised of three bars. In order from left to right, the bars represent YKD,

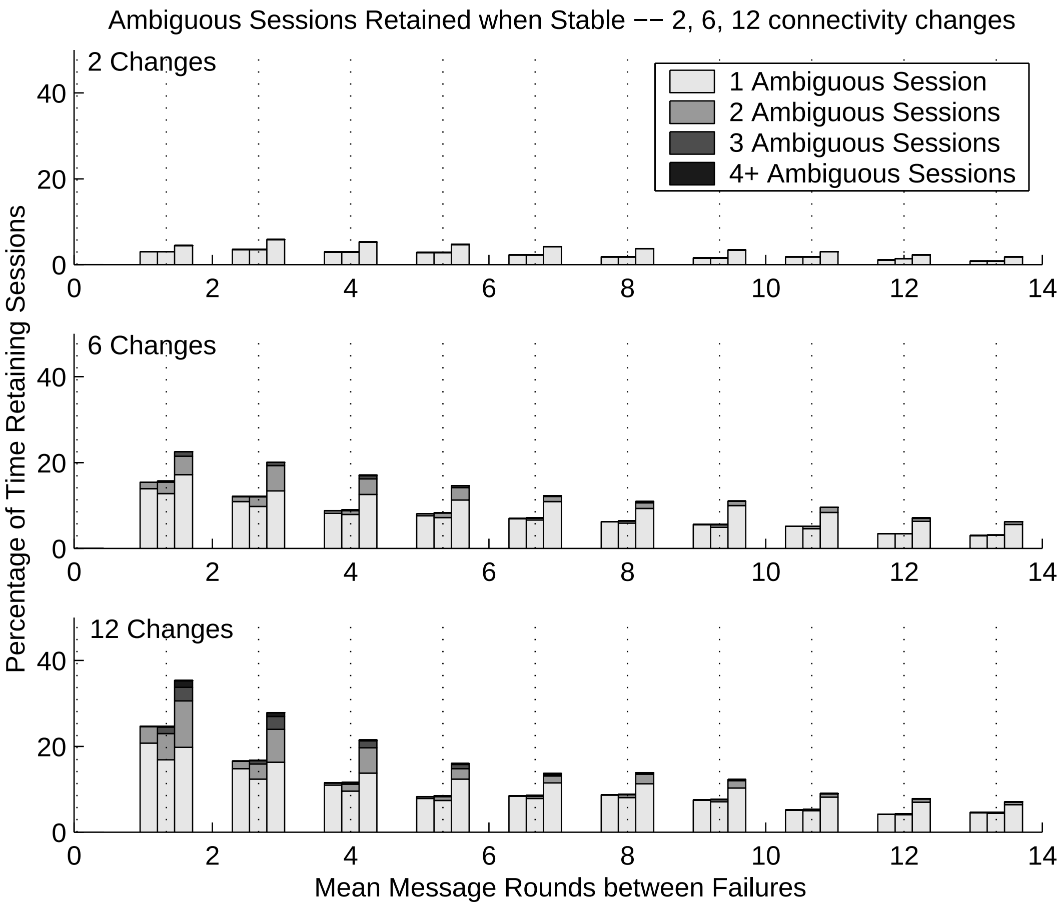

Figure 4-7: Ambiguous sessions with YKD, unoptimized YKD, and DFLS.

"Stable State" – measuring how many ambiguous sessions are retained when the algorithm has completed a run.

unoptimized YKD, and DFLS. The total height of the bar indicates the percentage of the time in which ambiguous sessions were retained. The bar is further divided into blocks, which indicate the actual number of ambiguous sessions retained. The bottom block represents a single retained ambiguous session, the second – two and so forth.

The most striking phenomenon observed is how few ambiguous sessions are retained. The theoretical worst-case number of ambiguous sessions 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 sessions retained never exceeded 9 for

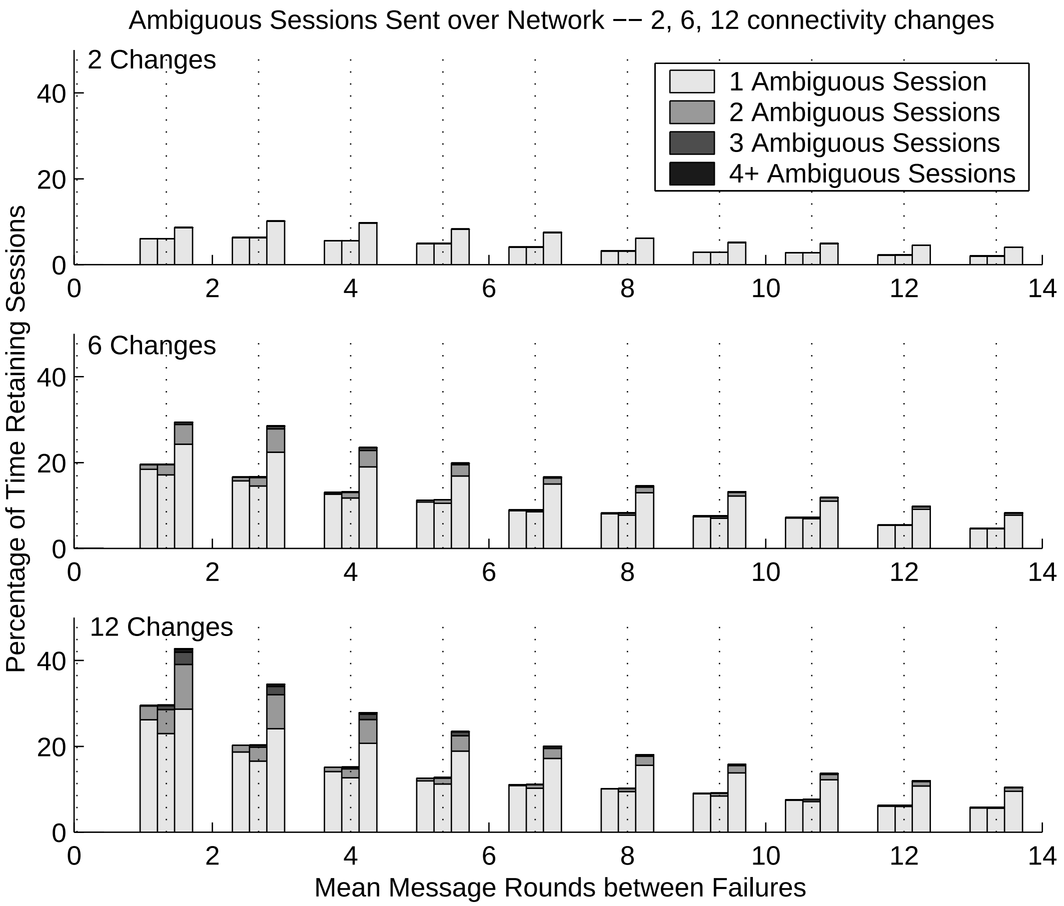

Figure 4-8: Ambiguous sessions with YKD, unoptimized YKD, and DFLS.

"In Progress" – measuring how many ambiguous sessions are retained during the algorithm's run, hence how many it must transmit across the network.

DFLS, and never exceeded 4 for YKD. The number of retained ambiguous sessions was dominantly zero. This demonstrates how unlikely the worst-case scenarios truly are.

This is primarily relevant to message size. Figure 4-8 is indicative of the number of ambiguous sessions retained at times when the algorithm must broadcast those ambiguous sessions to the other processes in the new view. The size of that broadcast message is directly related to the number of ambiguous sessions. The fact that the number of ambiguous sessions retained is generally quite low means that the broadcast message size is also fairly stable.

Please note that at the conclusion of a successful run, none of the algorithms retains any ambiguous sessions at all. Therefore, the bars are higher for DFLS simply due to the fact that succeeds less often, 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 sessions, on average.

Chapter 5

Conclusions

We have compared the availability of four[1] 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., [7, 10]).

We have shown that the number of processes that need be contacted and the number of message rounds required in order to resolve past ambiguous attempts significantly affect the availability. This is especially true as there are more connectivity changes, and as these changes become more frequent. The 1-pending and MR1p algorithms degrade drastically as the number and frequency of connectivity changes increase. In highly unstable runs with cascading connectivity changes, they are even less available than the simple majority algorithm. For 1-pending, this is because it sometimes requires a process to hear from all the members of its retained ambiguous session before progress can be made. MR1p does not have such a strong restriction, but the number of message rounds it requires to resolve an ambiguous session and form a primary is prohibitively high.

In contrast, the YKD algorithm [12] 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

[1] Again, we do not consider unoptimized YKD here, since its availability is equal to that of YKD.

the algorithm highly appropriate for deployment in real systems with extensive life spans.

The DFLS algorithm [5] degrades as gracefully as the YKD algorithm. However, it is less available than YKD for all failure patterns. This illustrates the effect of the speed at which attempts are resolved on the availability.

We have also measured the number of ambiguous sessions typically retained by YKD, unoptimized YKD, and DFLS. All of them retain surprisingly few ambiguous sessions during their operation, especially considering that the worst-case performance for DFLS and unoptimized YKD is exponential in the number of processes. This means that the amount of memory required to run the algorithm and the size of the messages which must be broadcast can both be constrained well in normal operation – in runs with 64 processes, message sizes can typically be constrained to two kilobytes or less.

5.1 Future Work

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[2] in order to study additional algorithms and to compare them with those studied here.

We also recognize that other failure models and probability functions can be explored as well. For example, we have not demonstrated algorithms' availability if one of the processes from the original view crashes, nor did we use anything other than a uniform probability distribution. We also did not take message size into account when computing availability. Researchers may also wish to use our implementation of YKD and its variants as a basis for developing and running other tests on the existing algorithms.

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

Bibliography

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

[2] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Transis: A communication subsystem for high availability. In *22nd IEEE Fault-Tolerant Computing Symposium (FTCS)*, July 1992.

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

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

[5] 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.

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

[7] 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.

[8] 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.

[9] 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.

[10] 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.

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

[12] 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.