Ideal Stabilization in a PIF Chain

Jordan Adamek
Department of Computer Science
Kent State University
Kent, OH 44240, USA
jadamek2@kent.edu

Department of Computer Science

Kent State University

Technical Report: TR-KSU-CS-2011-02

Abstract. I test the practicality of ideal stabilization. Self-stabilization, or classic stabilization, is a specification property which guarantees that a program will eventually satisfy its specification regardless of transient faults or lack of initialization. Classic stabilization loses some appeal as a new standard for fault tolerant distributed systems due to unattractive features such as poor composition and uncontrolled behavior during fault recovery. Ideal stabilization is a special case of stabilization which eliminates these two disadvantages. However, ideal stabilization has mostly been discussed in theory and needs practical testing. I test this property in a real program with real faults to observe the finer details of its behavior in a practical setting. Specifically, I create a simulation engine to test the behavior of a program with the propagation of information with feedback (PIF) specification. This specification is proven to ideally stabilize, and is a well-known model for message passing and clock synchronization in distributed systems. The PIF specification usually comes in the form of a tree, but for simplicity I reduce this tree to a chain. I present this program with initial faults as well as totally random initial states, and observe its behavior in regards to fault tolerance during computational operation.

Introduction

Self-stabilization (or just stabilization) is a specification property which guarantees a program will eventually satisfy its specification regardless of initial state or presence of transient faults during computation. A transient fault is an error that is introduced during program computation as opposed to a major fault, such as a hard drive or motherboard crash. The property of self-stabilization, which was first brought to the attention of the computing community in 1974 by Edsgar Dijkstra [1], presents a great deal of potential to be a pinnacle of program specification design around which fault tolerance in future systems may center. However, stabilization is sometimes seen as too strong or optimistic a property, as it presents no control over the initial state leaving it arbitrary and open for corruption. As discussed in [2], these algorithms are inherently non-terminating, meaning the program will continue to function even after a fault has occurred leading to widespread corruption during computation. This unpredictability, as mentioned in [6], further tarnishes the image of stabilization as a fault tolerance standard. Lastly, research into stabilization may be seen as unreliable in some cases due to unstated limitations on the models observed, and a need for a generalized technique to stabilization is apparent.

Stabilizing programs often have complicated structures to give the programmer more control over the behavior of transient fault correction. Usually this comes in the form of layers or other combinations of substructures which seek to localize a fault and correct it before it passes to the next substructure. The behavior of these programs can be totally unpredictable, and may not guarantee the program adhere to the recovery operations of the specification.

Ideal stabilization is presented by [3] as a special case of classic stabilization which eliminates these two unattractive features. Ideally stabilizing programs always obey their specification during fault correction and handling, and have no constraints on their composition. Ideal stabilization also gives the programmer full control over fault correction behavior. Ideal stabilization could therefore be the avenue down which stabilization makes its stand as the standard approach to fault tolerance in distributed systems, as it eliminates the main factors which deter from the appeal of other forms of stabilization.

The propagation of information with feedback (PIF) is a program specification which is a well-known model for distributed system algorithms. This specification is usually in the form of a tree. For simplicity, this tree was made to have no branches, effectively reducing it to a PIF chain. This PIF chain easily simulates message passing and clock synchronization protocols, two very pertinent topics in distributed system algorithmic research. This PIF chain is also proven to ideally stabilize, making it a perfect test subject by which to observe the behavior of ideal stabilization in distributed system algorithms.

Related Work. Snap-stabilization presents a quicker and more efficient means of stabilization. As explored in a propagation of information with feedback (PIF) tree in [4, 5], snap-stabilization presents an approach to stabilization which is both time-optimal and optimal in the number of states taken to correct any faults. However, this is still a delay, and during this delay, the program may completely disregard the specification in order to achieve predicate legitimacy in optimal time.

Our contribution. Ideal stabilization is still a very new area of research in the world of distributed system computing, and needs practical testing against real programs with real faults. This paper will cover an experiment tested an ideally stabilizing PIF chain in a simulation engine against real initial and transient faults. This paper will analyze the behavior of this ideally stabilizing program, and seek to draw a better understanding of the real and practical implications ideal stabilization has on the area of distributed system computing.

Notation

This section will cover all terms used throughout the rest of the paper. Even to readers thoroughly familiar with the subject, it is crucial to read this section and understand these terms, as they will be used explain all results and conclusions drawn from the experimental data.

Programs, processes, and states. A program is built to satisfy a specification and contains any number of processes. These processes may have any number of variables, whose values range over a fixed domain. The state of a process is the current assignment of values from that domain to the variables within the process. This is called the *process state*, and the assignment of a value to all variables of each process makes up the *program state*. The *state universe* describes all possible sequences of program states that may occur. The specification the program is built to satisfy describes the total set of satisfactory program states. All sequences of program states within the state universe that satisfy the specification are *allowed* by the specification, and are *disallowed* otherwise. If the allowed states of a specification include all states within the state universe, the specification is *ideal*.

Actions. A process contains any number of *actions*. An action *transitions* the current program state to another state in its state universe. This transition is called an action *execution,* and is instigated by the *command* of the executed action. The *command* of an action is a sequence of variable assignments and branching statements as defined by the process. The *guard* of an action is a predicate, or rule on the program state, which describes the enablement of that action. If the guard is adhered to, and thus evaluates to true, the action is considered *enabled*. The action is *disabled* otherwise. Only enabled actions may be executed at any program state. A maximal sequence of action executions, or transitions of the program state from one to another, is called a *computation*. By maximal, this means that the computation ends only when no actions of any process are enabled at a given program state, and may also continue on infinitely rather than being terminated abnormally, such as through a terminating command or major fault.

Predicates, legitimacy, and stabilization. A state *predicate* is a set of rules or Boolean expressions over a program state. If the predicate evaluates to true for that program state, the state *conforms* to the predicate. If every state of every computation conforms to the predicate, provided the computation begins in a state that conforms to the predicate, the predicate is considered *closed*. If every computation of a program starting in a state that conforms to a closed predicate maps to an allowed sequence, that predicate is an *invariant* of the program. If a program state conforms to the invariant, it is a *legitimate* state, and *illegitimate* otherwise.

A program is defined *stabilizing* if, from any arbitrary initial state in its state universe, it will always eventually reach a legitimate state. Let it be noted the definition of an invariant guarantees that once the program has reached a legitimate program state, it will never map to an illegitimate state thereafter.

PIF specification. The propagation of information with feedback is a program specification which can be used to simulate message passing or clock synchronization. This specification is usually in the form a rooted tree with any number of branches. For this experiment, the PIF tree was reduced to only a trunk with no branches, effectively

reducing it to a PIF chain, which we visualize as a horizontal chain. Processes to the right are *descendants;* processes to the right are *ancestors*. This chain contains a leftmost *root* process, a rightmost *leaf* process, and *N* number of *intermediate* processes which all have one state variable called *st*. For the intermediate process, the variable st has a domain consisting of the following values: requesting *rq*, idle *i*, and replying *rp*. For the root process, st cannot be replying, only idle or requesting. For the leaf process, st cannot be requesting, only idle or replying.

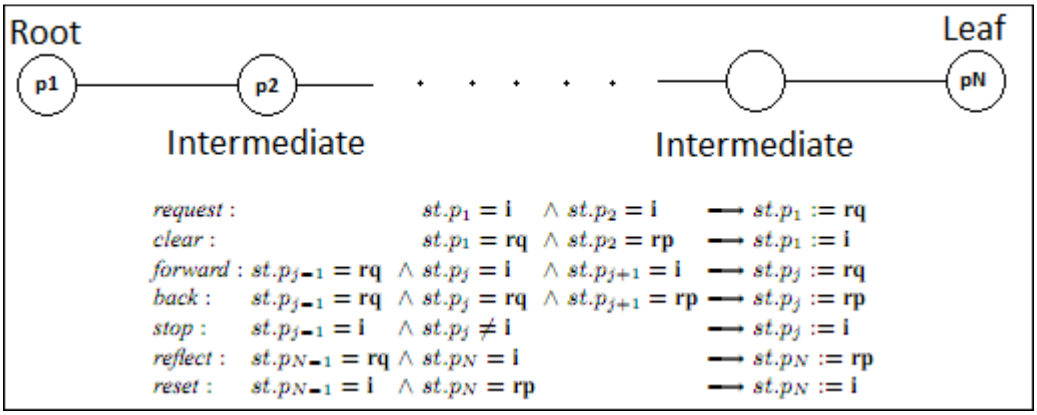

Figure 1: PIF Chain visualization and list of actions in the form: *(name) : (guard)* ⟶ *(command)* belonging to all processes. Here, j is an integer between 2 and N-1 representing the process index. The root process p_1 contains the actions *request* and *clear*. The leaf process p_N contains actions *reflect* and *reset*, and all intermediate process p_j (j = 2, N-1) contain actions *forward*, *back*, and *stop*.

This program specification has two predicates, RP(k) and RQ(l, m), which can be proven as a collective invariant of the program. By collective, I mean that the invariant is actually the expression: *RP(k)* ∨ *RQ(l, m)*. A legitimate state is not defined as one that conforms to both predicates at once, but may be equal to just one or the other. Also, being an invariant, all possible state transitions from a state described by these two predicates will always transition to a state conforming to either one or both predicates. This invariant is what the PIF chain ideally stabilizes to. This means that any action executed on any arbitrary program state in the chain's state universe will always bring the program closer to conforming to one predicate or the other if it does not conform already.

The first predicate is called RP(*k*), where *k* can be any process from 1 to N-1. This predicate describes a program state in which all processes equal to or less than *k* are requesting, and all remaining processes up to process N are replying.

The second predicate is called RQ(*l*, *m*), where *l* is between 0 to N-1, and *m* is from *l*+1 to N. This predicate describes any program state in which all processes equal to or less than *l* are requesting, all process greater than *l* and equal to or less than *m* are idle, and the rest are replying.

Keep in mind that *l* can be equal to 0, which describes a program state in which no processes are requesting, only idle or replying. Also, *m* may be equal to N, which describes a program state in which no processes are replying, only idle and requesting. Taking both of these observations into account, if *l* = 0 and *m* = N, we see that

a program state in which the state variables of all processes are all set to idle actually conforms to predicate RQ(l, m), and therefore conforms to the invariant qualifying it as a legitimate program state. This totally idle legitimate state was a key point in the experiments conducted in this paper.

Experimental Setup

I wrote a simulation engine in Ruby Script in which I coded the structure of the propagation of information with feedback (PIF) chain and the definitions of all its parts and predicates. The chain itself was coded as a Tree class containing an accessible data array of size *SIZE* and a single integer variable *size.* I defined three object classes: Root, Leaf, and Intermediate to simulate the three different types of processes in the chain. Within these three classes I defined the state variable and actions belonging to that class. For the Intermediate class, I defined another integer variable *index* which kept track of the processes index number (*index* = 2, SIZE - 2). I then defined the guard and command of each action separately. I included an *enabled?* method in each class to return the guard of an action for that process at a given program state. I then defined another method *actions* which would invoke *enabled*? for all *available* actions of that class, and return an array containing all enabled actions (for which *enabled?* returned a value of ***true)***. I lastly defined a method called *flip* which set the state variable of the process to a non-idle value (replying or requesting).

For the tree's initialization, I added an object of class Root to the data array. Then, I wrote a loop in which the content of the data array at index *i* was set equal to an object of class Intermediate with incrementing index *i*. Lastly, I added the an object of class Leaf to the end of the array after the loop finished execution. I set this loop to run from *i* = 1 to SIZE – 2. Lastly, I included two accessor methods for the tree's data array and size variables.

I incorporated the invariant predicates RP(k) and RQ(l, m) as actual Boolean methods which returned the conformity of the program state against the predicates at one specific value for each of the three parameters: k, l, and m. I next defined *RP?(k)* and *RQ?(l, m)* which acted as recursive methods invoking RP(k) and RQ(l, m) for all possible values of k, l, and m. If and when the methods RP(k) or RQ(l, m) returned ***true*** for a given value of k or set of values l and m, the recursive method would return a total value of ***true***. I lastly defined a method called *legit* which returned the Boolean expression: *RP?(k) OR RQ?(l, m).* If both RP?(k) or RQ(l, m) returned a value of ***false***, the method *legit* returned ***false***, otherwise it returned a value of ***true***.

I then wrote a method called *run(c)* which simulated the creation and computation of a PIF chain. In this method, parameter *c* represents the index of the current computation. I ran multiple computations for each experiment, and it was thus necessary to use an index to organize the recording of these results into numbered logs in the form of text files. In this method, I initialized the chain to be in a completely idle state, which as pre-mentioned is actually a legitimate state and was used as a sort of *ground state* from which the PIF chain was considered clean. Next, I invoked a *perturb* method. This method randomly selected *FLIP* number of process from the tree's data array, invoking the flip method of each. This set each process's state variable to a non-idle value, *perturbing* that process. These initial perturbations, which represented initial transient faults, most often had the effect of causing the initial state to now be illegitimate (as a fault in the initial state of a program would normally do).

This ground legitimate state is defined in my experiments as a program state in which all state variables of all processes are idle. Faults at initialization are defined as setting the state variable of processes to non-idle values. This led to an interesting implication which became the basis for two of my five conducted experiments.

Consider a totally idle program state. A certain number of perturbations are made on that state, flipping the state variable of a number of currently idle processes to non-idle values. To return to the ground state, and thus a legitimate state, the program need only flip those processes' state variables back to a value of idle. Using this line of reasoning, let us define any action whose command sets the state variable of its process to a value of idle as a *stabilizing* action, as it helps return the program state to the ground legitimate state. If we bias the program to only execute stabilizing actions, the program would reach legitimacy in a number of states equal to or less than the number of faults present at initialization, making for highly efficient fault correction. However, the program would in this instance lose all functionality. Thus, I define a *stabilizing-action bias* as the percent-ratio by which, for a given state transition, only stabilizing actions were considered for execution over all others.

Here, I illustrate method run(c) in pseudo-code to provide a better visual of my coding structure:

```
return if legit
    iter = 1
    while(iter <= N)
      if rand(99) < FUNC
        #if the program chooses to execute a stabilizing action:
        for j in 0...SIZE
          #if an action is enabled for process j, that process is included in
          #enabled array
          enabled.add(j) unless $tree[j].enabled_actions.empty?
        end
        process = $tree[enabled.pick]
        action = process.enabled_actions.pick

      else
      #executes a stabilizing action
        for j in 0...SIZE
          #only process which have a stabilizing action enabled are added to
          #the enabled array
          enabled.add(j) if $tree[j].enabled_actions.include?("clear")
          enabled.add(j) if $tree[j].enabled_actions.include?("stop")
          enabled.add(j) if $tree[j].enabled_actions.include?("reset")
        end
      end

      execute_action
      #ends the computation if the state is legitimate
      if legit
         legit_point = iter
         break
      end
      iter += 1
    end
end
```

Figure 2: Computation pseudo-code in mock Ruby Script. In this illustration, the PIF chain has already been initialized as the object *$tree* .The initial program state is set either to completely idle or a random set of values. *pick* is an array method which selects an element from that array at random. *FUNC* is a constant which represents the inverse of the stabilizing-action bias. When this value is equal to 100, there is 0% bias and the program always considers all enabled actions when non-deterministically (in this case randomly) choosing a process and action to execute. When this value is equal to 0, there is 100% bias and the program behaves in the pre-mentioned maximally efficient but completely nonfunctional manner.

This computation algorithm first determines if the current program state is legitimate. Recall that once the program reaches a legitimate state, it will never transition back to an illegitimate state. Therefore, once the program state is evaluated as legitimate, the computation is terminated and the program records the current state index *iter* as the number of state transitions required to reach legitimacy. For arbitrary initial states that were already found to be legitimate, the number of state transitions made was recorded as zero. The algorithm then randomly selects an enabled action and its process. That action is then executed, and once again the program state is checked for legitimacy. This procedure is repeated until the program reaches a legitimate state.

I created a loop to repeat this computation algorithm one-hundred times, each time logging the results of the computation as the number of state transitions to legitimacy in the results text file. I did this to ascertain a more accurate result in the form of an average rather than a single value for a single computation which would provide little information to us.

For the first experiment, *I* set the size of the chain to 20 processes, and set the stabilizing-action bias to 0. I initialized the PIF chain as totally idle, and incremented the number of initial faults from 1 to 20 for each set of 100 computations. The results of each set were recorded in a folder indexed according to the number of perturbations made at initialization, and the averaged value was recorded in a separate text file which housed the averaged results of all twenty computation sets. These averages were then transferred to an Excel spreadsheet in both chart and tabular form, which you may observe in *Figure 3* and *Table 1* of the *Results* portion in this paper**.**

In the second experiment, I tested a chain size of 10 processes with again no stabilizing-action bias present and a totally idle initial program state. Here, I incremented the number of perturbations made at initialization from 1 to 10. The results of this experiment are recorded in *Figure 4* and *Table 2* of the Results section in this paper.

In the third experiment, I implemented the stabilizing-action bias. I returned to a chain size of 20 processes, and set the stabilizing-action bias to 100%. For a fourth experiment, I set this bias to 50%. In both of these experiments, I retained a chain size of 20 and a totally idle initial state, incrementing the number of perturbations made at initialization from 1 to 20. Lastly, I averaged the number of states to legitimacy over one-hundred computations for a given number of perturbations as the result of each set. The results of the third experiment are recorded in *Figure 5* and *Table 3*, and the results of the fourth experiment are recorded in *Figure 6* and *Table 4* of the Results section in this paper.

In the fifth experiment, I kept the chain size set to 20 processes and set the stabilizing-action bias back to 0%. This time however, I did not start from a totally idle initial program state, and instead assigned a totally random initial state. No perturbations were made on the initial state, and thus had more sets of 100 computations been tested, no variable would have been changed between one to another. Therefore, it is reasonable to say that had I tested and recorded more 100 computation sets, the results would have all been similar to one another providing no further information, only marginally increased accuracy. The results of this experiment can be found in *Table 1* of the Results section in this paper under the row *Random*.

Results

For the first four of my five experiments, I recorded the resulting data in both chart and tabular form. In each chart, I varied the number of perturbations made on the initial state as the independent variable (the x-axis). For each value, I record the resulting average number of states to legitimacy over 100 computations as the independent variable (y-axis). I also include 95% Confidence Interval error bars in each chart.

In each table, the results of each experiment are treated as a population of 100 computations. The resulting average, standard deviation, and 95% confidence interval of each 100 computation set is recorded in the row pertaining to the number of faults present at initialization (perturbations made on the ground state). In *Table 1*, the row labeled *Random* refers to the fifth experiment in which I only tested a single set of 100 computations with a totally randomized initial state.

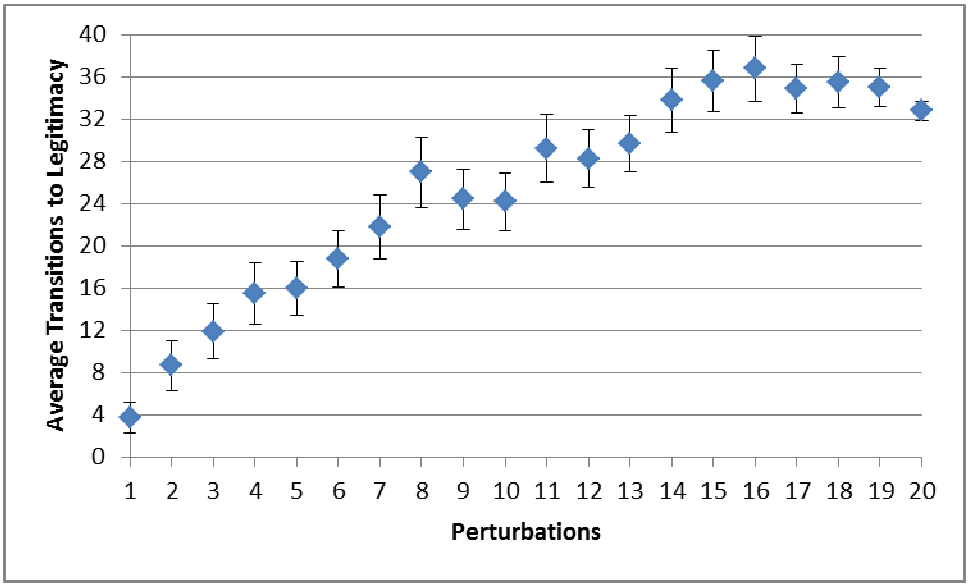

Figure 3: Number of perturbations vs. average states to legitimacy over 100 computations with confidence interval error bars. Here, the chain size is 20 processes and the stabilizing-action bias is 0%. Note the local maximum at 16 perturbations.

Number of Perturbations	Average States to Legitimacy	Standard Deviation	Confidence Interval (95%)
1	3.734	7.278	1.426
2	8.670	11.981	2.348
3	11.927	13.486	2.643
4	15.486	14.760	2.893
5	16.000	13.153	2.578
6	18.743	13.446	2.635
7	21.761	15.465	3.031
8	26.982	16.949	3.322
9	24.431	14.450	2.832
10	24.183	14.056	2.755
11	29.229	16.549	3.244
12	28.266	14.089	2.761
13	29.688	13.559	2.658
14	33.789	15.596	3.057
15	35.587	14.860	2.913
16	36.761	15.581	3.054
17	34.890	11.669	2.287
18	35.450	12.219	2.395
19	35.009	9.279	1.819
20	32.761	4.605	0.903
Random	28.850	13.796	2.704

Table 1: Experimental results for a chain size of 20 with 0% stabilizing-action bias. The results of the fifth experiment with random initial state are recorded here in the last row labeled *Random*.

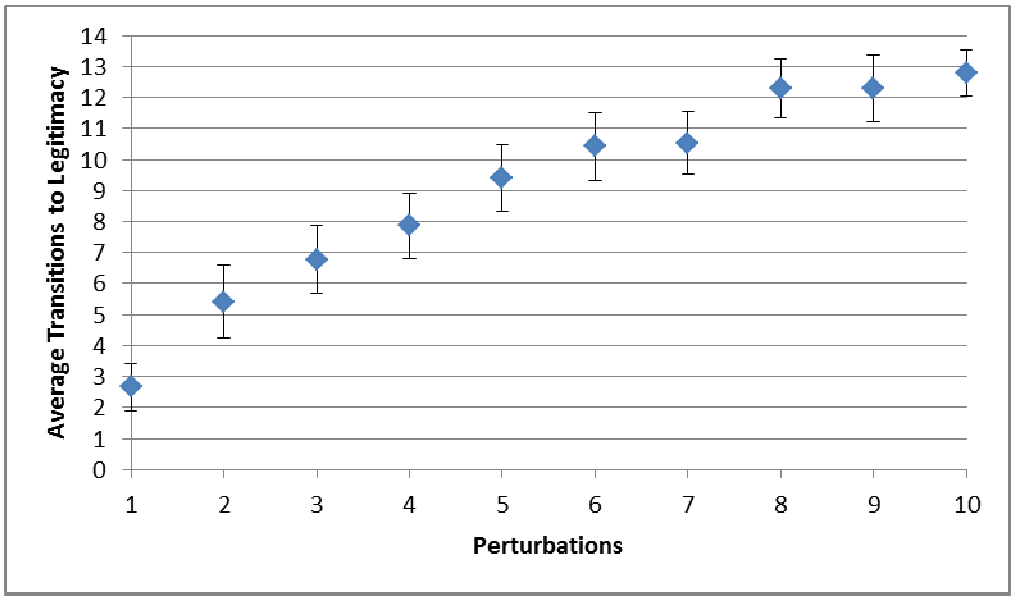

Figure 4: Number of perturbations vs. average states to legitimacy over 100 computations with confidence interval error bars. Here, the chain size is 10 processes and the stabilizing-action bias is 0%.

Number of Perturbations	Average States to Legitimacy	Standard Deviation	Confidence Interval (95%)
1	2.660	3.897	0.764
2	5.410	6.042	1.184
3	6.780	5.628	1.103
4	7.860	5.428	1.064
5	9.390	5.481	1.074
6	10.420	5.623	1.102
7	10.520	5.153	1.010
8	12.310	4.855	0.952
9	12.300	5.496	1.077
10	12.800	3.776	0.740

Table 2: Experimental results for a chain size of 10 with 0% stabilizing-action bias.

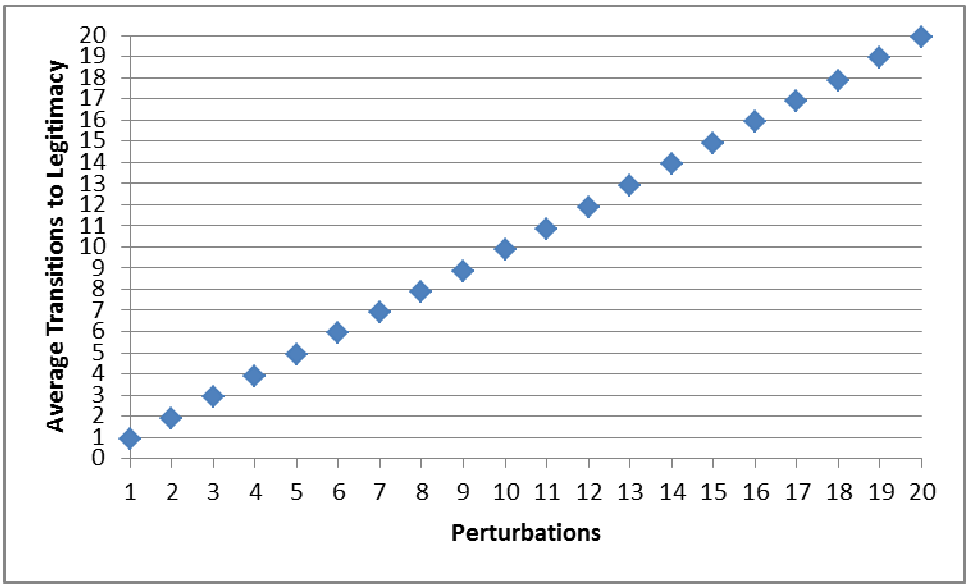

Figure 5: Number of perturbations vs. average states to legitimacy over 100 computations with confidence interval error bars. Here, the chain size is 20 processes and the stabilizing-action bias is 100%. This experiment tested the hypothetical maximally biased program discussed in the preceding sections.

Number of Perturbations	Average States to Legitimacy	Standard Deviation	Confidence Interval (95%)
1	0.930	0.255	0.050
2	1.880	0.354	0.069
3	2.900	0.300	0.059
4	3.860	0.347	0.068
5	4.900	0.361	0.071
6	5.910	0.377	0.074
7	6.890	0.343	0.067
8	7.900	0.332	0.065
9	8.830	0.448	0.088
10	9.880	0.325	0.064
11	10.870	0.439	0.086
12	11.850	0.384	0.075
13	12.870	0.336	0.066
14	13.920	0.337	0.066
15	14.870	0.365	0.072
16	15.900	0.332	0.065
17	16.920	0.306	0.060
18	17.870	0.416	0.082
19	18.930	0.255	0.050
20	19.910	0.286	0.056

Table 3: Experimental results for a chain size of 20 with 100% stabilizing-action bias.

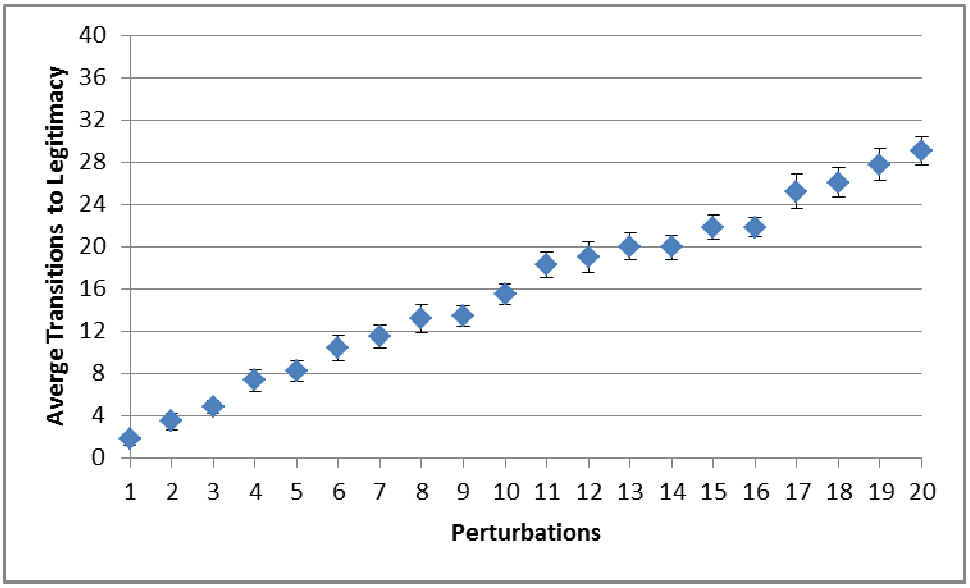

Figure 6: Number of perturbations vs. average states to legitimacy over 100 computations with confidence interval error bars. Here, the chain size is 20 processes and the stabilizing-action bias is 50%.

Number of Perturbations	Average States to Legitimacy	Standard Deviation	Confidence Interval (95%)
1	1.820	2.794	0.548
2	3.490	3.999	0.784
3	4.840	2.966	0.581
4	7.360	5.440	1.066
5	8.220	5.096	0.999
6	10.450	6.341	1.243
7	11.520	5.454	1.069
8	13.200	6.560	1.286
9	13.470	4.920	0.964
10	15.560	4.969	0.974
11	18.280	6.248	1.225
12	19.030	7.233	1.418
13	20.000	6.475	1.269
14	19.930	5.994	1.175
15	21.840	6.049	1.186
16	21.850	4.782	0.937
17	25.240	8.462	1.658
18	26.090	7.199	1.411
19	27.790	7.520	1.474
20	29.090	6.720	1.317

Table 4: Experimental results for a chain size of 20 with 50% stabilizing-action bias.

Data Analysis

The results of the five experiments conducted indicate that the behavior of ideally stabilizing programs adheres to the theory, though not exactly as predicted. I expected to see a linear relationship between the number of initial faults and the average number of state transitions made to reach legitimacy. However, the shape of the data for *Figure 3* indicates that this relationship is close to linear, but has a local maximum at around an 80% fault to size ratio (16 initial faults for a chain size of 20). Also, I note that a completely random initial state for a chain size of twenty processes did not result in the largest average number of transitions. For a chain size of 20, a completely random initial state yielded an average of about 28.8 state transitions as seen in *Table 1.* The average number of transitions to legitimacy for an initial finite number of faults greater than twelve exceeded this value. Also, the standard deviation measurements indicate that the program is least predictable at a finite number of faults closer to the median, or a 50% fault to chain size ratio. Once again, a completely random initial state did not yield the maximum result, and was actually exceeded by finite numbers of initial faults between 8 and 16. These two analyses suggest that a completely random initial state does not yield the largest disturbance in the programs behavior, or that said behavior is at its least predictable, but that there is actually a finite number of initial faults which causes the greatest perturbation in the program.

Experiment 2, which tested a smaller chain size of 10 processes, yielded results of a more linear relationship as seen in *Figure 4*. Here, no local maximum was observed. This experiment simulates a simpler program with a low chain size, and adheres to my predictions for the relationship between the number of initials and the average number of state transitions to legitimacy. This indicates that this predicted behavior may only hold true for small programs.

The other two experiments, which tested stabilizing-action bias, yielded perhaps the most predictable results of this paper. The results for Experiment 3, which tested a program with 100% bias, were precisely as I predicted: the average number of transitions to legitimacy was equal to or less than the number of initial faults. The results of the experiment testing 50% bias yielded a mix of Experiment 1 and Experiment 3, where the results were closer to linear and again produced no local maximum.

Conclusions

The behavior of a PIF chain under initial transient faults is only linear for very small chain sizes, or when the program is introduced with a stabilizing bias. Also, a completely random initial state does not produce the greatest perturbation in the program. A local maximum occurs which produces the greatest disturbance in the program at a number of initial faults around close to 80% of the chain size. The experiment in which this maximum occurred tested a chain size which was still relatively low at 20 processes. In all, the concept of ideal stabilization in a PIF Chain was proven to be a highly practical method for fault tolerance and handling. More investigation into its behavior is required, as the nature of the local maximum observed in the first experiment could have strong implications for the use of ideal stabilization as a model for fault tolerance and handling.

Future Studies. Future experiments of this kind should first and foremost test larger chain sizes, perhaps sizes of one hundred or one thousand processes. This would yield a more precise value for the -finite number of faults which produces a maximum disturbance. Also, the precision of future results could benefit from an increased computation resolution, extending the number of computations per data point to a few hundred or even one thousand. These results as well as the entire concept of stabilizing-action bias were centered on a specific definition of a transient fault as a non-idle process state value. Future experiments could use a different definition of initial fault which somehow incorporates all three possible state values: idle, requesting, and replying as an initial fault rather than only replying or requesting. Lastly, future experiments should also seek to test other types of ideally stabilizing programs, as this property shows great potential to become the central model for fault tolerance and handling in programming.

References

1. Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. Commun. ACM,17(11):643–644, 1974.
2. S. Dolev. Self-stabilization. MIT Press, March 2000.
3. Mikhail Nesterenko, Sébastien Tixeuil: Ideal Stabilization. AINA 2011: 224-231
4. Alain Bui, Ajoy Kumar Datta, Franck Petit, and Vincent Villain. State-optimal snapstabilizing pif in tree networks. In Arora [7], pages 78–85.
5. Sylvie Dela¨et, St´ephane Devismes, Mikhail Nesterenko, and S´ebastien Tixeuil. Brief announcement: Snap-stabilization in message-passing systems. In Principles of Distributed Computing (PODC 2008), August 2008.
6. Jerzy Brzeziński, Michał Szychowiak. Self-Stabilization in Distributed Systems - a Short Survey. Foundations of Computing and Decision Sciences, 25(1), 2000.
7. Anish Arora, editor. 1999 ICDCS Workshop on Self-stabilizing Systems, Austin, Texas, June 5, 1999, Proceedings. IEEE Computer Society, 1999.