2011

An Efficient Fault-Tolerant method for Distributed Computation Systems

Samaneh Navabpour
University of Windsor

An Efficient Fault-Tolerant method for Distributed Computation Systems

by

Samaneh Navabpour

A Thesis
Submitted to the Faculty of Graduate Studies
through the School of Computer Science
in Partial Fulfillment of the Requirements for
the Degree of Master of Science at the
University of Windsor

Windsor, Ontario, Canada

2009

An Efficient Fault-Tolerant method for Distributed Computation Systems

by

Samaneh Navabpour

APPROVED BY:

_____Jonathan Wu_____
Dr. Jonathan Wu
Department of Electrical and Computer Engineering

_____ Yung Tsin _____
Dr. Yung Tsin
School of Computer Science

_____ Jessica Chen _____
Dr. Jessica Chen, Advisor
School of Computer Science

_____ Dan Wu _____
Dr. Dan Wu, Chair of Defense
School of Computer Science

June 16 2009

Author's Declaration of Originality

I hereby certify that I am the sole author of this thesis and that no part of this thesis has been published or submitted for publication.

I certify that, to the best of my knowledge, my thesis does not infringe upon anyone's copyright nor violate any proprietary rights and that any ideas, techniques, quotations, or any other material from the work of other people included in my thesis, published or otherwise, are fully acknowledged in accordance with the standard referencing practices. Furthermore, to the extent that I have included copyrighted material that surpasses the bounds of fair dealing within the meaning of the Canada Copyright Act, I certify that I have obtained a written permission from the copyright owner(s) to include such material(s) in my thesis and have included copies of such copyright clearances to my appendix.

I declare that this is a true copy of my thesis, including any final revisions, as approved by my thesis committee and the Graduate Studies office, and that this thesis has not been submitted for a higher degree to any other University or Institution.

Abstract

Fault tolerance is one of the most important features required by many distributed systems. We consider the efficiency issues of constructing distributed computing systems that can tolerate Byzantine faults. The well-recognized technique is to introduce replicated computation and derive the correct results through a voting mechanism. While this technique is applied to each computation request individually, we believe that by considering multiple requests at the same time in a distributed environment, we can greatly improve its efficiency. This is based on the observations that computation requests may be ordered in a different way for computation at different nodes, and the verdict of the correct result for one request may imply the correct result for another request. We propose to exploit a suitable solution to improve the efficiency of the existing technique to avoid unnecessary computation and unnecessary message exchanges among distributed processes.

We consider Peer-to-Peer architecture which is one of the well-known and fast growing architectures for distributed systems. The voting mechanism however may be either centralized or distributed, which may raise different issues and lead to different solutions.

Dedication

I want to dedicate my master thesis to my loving family, for their endless support and unconditional love; Specially my mother, “Mahshid”, for sacrificing her happiness for mine and my sister, “Sanieh”, for being my single point of hope in times of despair.

Acknowledgements

I would like to take this opportunity to thank my supervisor, Dr. Jessica Chen, for her endless support, invaluable guidance, and consistent encouragements and patience. Without her help, I could have never come this far and accomplish the work presented in this thesis.

In addition, I would like to thank my internal and external readers, Dr. Yung H. Tsin and Dr. Jonathan Wu for their valuable and helpful comments and suggestions on this thesis.

I want to specially thank our graduate secretary, Mandy Turkalj for her consistent help and endless smiles even on cloudy days.

In the end I want to take the time to thank my loving family. I want to thank my father, Hamidreza, and my mother, Mahshid, for their endless support and sacrifices to help me achieve my Master degree. I want to thank my sister, Sanieh, for being the shoulder I could cry on, and my brother, Ali, for constantly reminding me that nothing is worth sacrificing my peace of mind.

Table of Contents

List of Tables

List of Figures

Chapter 1. Introduction

From a consumer's point of view, an ideal system should always be *reliable* and *available* [32].

- Reliability is the likelihood that a system will remain operational despite of failures for the duration of a mission. Very high reliability is most important in critical applications such as the space shuttle or industrial control, in which failure could mean loss of life [32].
- Availability expresses the fraction of time a system is operational despite failures. It is important to note that a system with high availability may in fact fail. However, its recovery time and failure frequency must be small enough to achieve the desired availability. High availability is important in many applications, including airline reservations and telephone switching, in which every minute of downtime translates into significant revenue loss [32].

An absolutely reliable and available system is very often impossible in practice, because customers will allow for only a certain amount of time and money to spend for this feature, restricting system developers with finite resources to achieve reliability. As a consequence, we as consumers of technology and computers have regularly encountered failures, either in the form of a software crash, a hardware failure, or a power loss, etc. Systems may fail for many reasons [32]: (i) It may have been specified erroneously, leading to an incorrect design; (ii) It may contain a fault that manifests only under certain conditions that were not tested; or (iii) It may fail due to some unexpected environment change. In some cases these failures are just annoyances while in some other cases, they result in significant loss. The latter has become more common than the former as today's societies are becoming more and more dependent on computer systems.

Over the years, the industry and academia have presented and used various techniques to best approximate an ideal system with limited resources. Such techniques and technologies have been categorized as *fault tolerant techniques*. Fault-tolerance refers to numerous issues concerning different aspects of the development, deployment, and

maintenance of a system, the two most common ones being *reliability* and *availability* [32].

Fault tolerant techniques are generally designed specifically for two types of systems: distributed and centralized. A major difference is that in distributed systems individual components may fail without affecting the functionality of the entire system. This very characteristic complicates the design and implementation of fault tolerant techniques for distributed systems. For this reason, the development of effective and efficient distributed fault tolerant systems has been the focus of many pieces of research work. It is also the topic of this thesis.

A fault tolerant distributed system can be defined as [30]:

1. a system whose behavior is well-defined in case of component failure;
2. a system capable of masking the fault in case of component failure.

Component failures are typically categorized as [30]:

1. crash or omission failure: When a component does not respond to a request or does not show any sign of liveness in a certain period of time.
2. timing failure: When a response is correct but untimely: either early or late.
3. a byzantine failure: When the response to a request is incorrect or the component's functionality deviates from its specification. We will be considering this type of faults in this thesis.

When designing a fault tolerant method to mask faulty behavior, it is important to see what faulty behavior the systems are capable of handling. Furthermore, since the recovery of the system is dependent on the likely failure behavior, one has to extend the specification of the nodes to incorporate their failure behavior (failure semantics) which can usually be specified in the same way as the specification of the semantics of the normal behavior.

One of the most common methods used to tolerate faults in distributed systems is *replication*. Two types of replication techniques are commonly used in distributed systems:

1. *temporal replication*: In this method, a single component executes a job/computation multiple times. When a component fails to complete a computation correctly because of any of the failures mentioned above, it will be required to do the computation some more times until it completes the computation correctly.
2. *spatial replication*: In this method, components of the systems are replicated (with possibly different versions of it) through out the system. The replicas can reside on a single physical machine or can be spread out among a number of physical machines. Note that a component in the system is logically mapped to a single *node* despite its physical location. Here, a computation is executed by a number of components, either in parallel or sequentially, until a correct result for the computation can be achieved. In this thesis, we will consider *spatial replication*.

Although with a powerful replication manager, *availability* can be guaranteed to a high degree, replication alone cannot provide *reliability*. The methods designed to achieve *reliability* can be put into two groups: *consensus/fault detection methods* and *fault detection/recovery methods*.

The first group aims at finding the correct result of a computation from among a group of variant results presented from different *replicas/redundant computation executions*. Consequently, when the correct result is found, faulty results are discarded and the faulty nodes which produced these results are detected and guided towards recovery. The most common aspects in this category include:

- *a voter*: the results from the replicas/redundant computations are analyzed by a *voter*. Each voter has a *policy* which defines which results are correct and which are not. This *policy* is dependant on the replication type, components, computation type, etc. of the system. In this case, the component(s) which have returned a result other than the correct one can be detected as faulty.

- *comparison of messages*: faulty nodes are detected via the messages they send. When a node shows different behaviors and sends out different messages for the same situation, this node is detected as faulty. This method is typically used in systems with *spatial redundancy* to find the correct result of a computation.
- *testing*: a testing method is capable of detecting faults in a node. The testing method on a node is designed based on the semantics of the computations this node execute.

The second group does not consider isolating faulty behavior. Instead, they focus on detecting faulty behavior in the system and dictating recovery guidelines for the faulty node. *Testing* and *comparison of messages* are part of this category as well.

In this thesis, we consider the *consensus/fault detection methods.* Our focus is to advance the existing voter techniques.
The basic tasks to achieve fault tolerance include:

1. *replicate components*: to replicate a same component throughout the system.
2. *consensus/fault detection*: to find the correct result of a computation, detect the faulty nodes, and discard incorrect results.
3. *recovery*: having faulty nodes detected, recovery steps are followed and the nodes are denied from further activity in the system until they are recovered.

The application of the above techniques is slightly different in stateful and stateless distributed systems. For stateful systems, if different non-faulty components/nodes execute a same set of requests in different orders, their states will deviate from each other, leading to different results. As a consequence, guaranteeing that all non-faulty nodes execute the same set of requests in the same order is vital for providing fault tolerance. We assume that the components of each node are stateless.

It can be seen that fault tolerance introduces considerable overhead in terms of the cost of resource requirements and computation time. Although the owners and consumers of a system desire for high degree of fault tolerance, they both can only accept a limited

overhead. Therefore designing and implementing an efficient fault tolerant method is of great importance. The overhead of fault tolerant methods is determined mainly in terms of

- the cost of the addition resources required,
- the additional time consumed for computation and analysis.

The cost of additional resources and computation time can be reduced in different ways:

- reducing the number of messages passed around, so that the cost of the needed network bandwidth can be reduced. When fewer messages are passed around, the system requires less bandwidth to function. Therefore the system becomes not only less expensive, but also more scalable and more efficient in wide area networks.
- reducing the number of replicas, so that the amount of hardware needed for the replicas to reside on can be reduced. When the number of replicas increases, the hardware required for the replicas increases as well. Therefore when fault tolerance can be guaranteed with a smaller number of replicas, the cheaper fault tolerant system can be developed while keeping scalability at the same time.
- reducing the time complexity of the computation and analysis, so that we will need less computation resources (CPU units, I/O devices etc.). When the time complexity of the computations is reduced, we consume fewer amounts of CPU units within each node. As a consequence, a more scalable and less time consuming fault tolerant system can be achieved.

This thesis aims at introducing an efficient distributed fault tolerant computation system, where the components are stateless. With respect to efficiency, we reduce the cost of additional recourses and the additional time consumption by:

1. reducing the number of computations and analysis,
2. reducing the time complexity of the computation and analysis

When a node is faulty, the majority of its computation results are incorrect. We assume that when a node becomes faulty, all of its computations are incorrect. Under this assumption,

1. when a node completes a computation incorrectly, we can conclude, without further computation, that all the results of the computations after this point (until its latest recovery) are incorrect.
2. when a node completes a computation correctly, we can conclude, without further computation, that all the results of the computations before this point (after its most recent recovery) are correct.

Making use of these observations, we further improve the efficiency of the existing methods by reducing the time complexity to find the correct results and reducing the number of computations needed to detect faulty nodes.

The continuing of this thesis is divided into the following Chapters: In chapter 2 we present the related works which have focused on developing efficient distributed fault tolerant systems. In chapter 3 we present the problem which has motivated our work. In chapter 4 we present our solution and algorithm. In chapter 5 we present the evaluation of our solution and in chapter 6 we have present our conclusion and future work.

Chapter 2. Related Work

In the following, we give a brief overview of the groundbreaking work in optimizing fault tolerant distributed systems.

The existing work on optimizing fault tolerant methods in distributed computation systems mainly focuses on the optimization of the following three aspects [18]:

- the protocol which orders the requests in stateful fault tolerant systems;
    - the method which finds the correct result of a request;
    - the method which detects faulty nodes.

Since the focus of this thesis is in *stateless* distributed systems, we are not concerned with the *ordering of the requests*. In *stateful* distributed systems, on the other hand, developing an efficient protocol for the *ordering of the requests* is of great importance. The most widely used protocol for establishing similar ordering of requests in all the nodes is Castro's BFT protocol [7]. The BFT ensures that:

1. all non-faulty replicas execute the same requests in the same order while less than one third of the nodes are faulty (have crashed or have byzantine behavior);
2. a node (user) will always choose the correct result for a request.

BFT places the replicas in a hierarchy, which is referred to as a *view*. In a *view*, there is a special replica serving as the *primary* and the others are *backups*. The structure of a view can change when the primary is found faulty.

BFT has three phases to atomically multicast requests to the replicas in a view. The three phases are *pre-prepare, prepare,* and *commit*. The *pre-prepare* and *prepare* phases are used to totally order requests in one view. The *prepare* and *commit* phases are used to ensure that requests are totally ordered when the structure of a view changes. In the *pre-prepare* phase, when the primary receives a request q from a node (user), it assigns a sequence number n to q and multicasts the sequence number to the backups in a message

called *pre-prepare*. When a backup receives the *pre-prepare* message, it enters the *prepare* phase by multicasting a *prepare* message to all the other replicas. The *prepare* message denotes that the backup has agreed to assign sequence number n to q in the current view. In the next step each replica waits until it has received $2f$ matching prepare messages regarding sequence number n and request q. Here f represents the maximum number of faulty nodes. At this point a replica is assured that a group of $2f$ replicas have agreed to assign number n to q in the current view. To this point BFT ensures that all non-faulty replicas have the same total ordering of requests in the same view, but it can not ensure that all nodes have the same total ordering of requests across view changes. The *commit* phase solves this problem. Each replica multicasts *commit* messages to notify others that it has completed the *prepare* phase. Then each replica collects messages until it has a group of $2f+1$ *commit* messages regarding sequence number n in a view v. At this point it is said that the request is committed and safe to be executed. The *commit* phase guarantees that there is a group of $2f+1$ replicas which know that a group of $2f$ replicas have accepted to assign number n to q in a certain view and are ready to execute q.

The view change protocol provides liveness by allowing the system to make progress when the primary fails [7]. When a backup suspects the primary of the current view v to be faulty, it enters view $v+1$ and multicasts a view change message to all replicas. Replicas collect view change messages for view $v+1$ and send acknowledgments for view $v+1$ to the new primary. The new primary collects view change acknowledgments and the view change happens when the new primary has $2f+1$ view change acknowledgement messages. In the end Castro et al proved that their protocol preserves safety and liveness. In addition they noted that their method only works when $3f+1$ replicas are active in the system and their method suffers from excessive message passing.

Amir et al presented a more efficient request ordering protocol [4] compared to BFT [7] by reducing the message overhead of the BFT protocol. In this method, nodes are

grouped into *sites*. Each site can have at most *3f+1* nodes. Here *f* represents the maximum number of faulty nodes. In each site one node is chosen as the representative to conduct the three phases of the BFT protocol within the site. In addition, one site is chosen as the leading site to achieve total ordering of the requests among all the sites using the BFT protocol. When the representative of a site or the leading site shows faulty behavior, a new node and a new site is assigned to these positions respectively. The changes made to these positions are similar to the view change in BFT. This method reduces the message complexity for wide area networks from $O(n^2)$ in BFT to $O(s^2)$. Here n is number of nodes, s is the number of sites and $s<<n$. Consequently their method increases the system's ability to scale.

In a more advanced work, Amir et al presented a new protocol (Blink) used between the sites to enhance performance in wide-area systems [5]. This new protocol establishes a reliable virtual communication link between the sites. This communication protocol discards the need for redundant message sending and allows a site to consume approximately the same wide-area bandwidth as a single physical node. The advantages of this method compared to their previous work [4] are: (i) The use of BLink increases performance by decreasing the number of redundant message passing in comparison to their previous work which typically requires $(f + 1)^2$ redundant message sends; (ii) The new system achieves high performance in terms of time complexity and outperforms their previous work by a factor of 4 when running in a system with the same number of faulty nodes. The problem of this method is that it exposes weak nodes to the malicious ones and does not provide confidentiality.

In addition to Amir's work [4,5], Kotla et al also presented a method [19] which optimizes BFT [7] by reducing the number of messages sent and decreasing the time consumed to find the correct result of a request. Unlike Amir's [4] work, replicas respond to a user's request without first running the three-phase BFT protocol. Instead, they

optimistically accept the order proposed by the primary and send the result immediately to the user. This causes a reduction in the number of message passing but the replicas can become temporarily inconsistent with one another. The users can detect these inconsistencies and help direct the replicas to converge to a single total ordering of requests. In their protocol, a user sends a request to the primary, then the primary forwards the request to the replicas, and the replicas execute the request after completing the *prepare* phase. Afterwards the replicas send their results to the user. Each message containing a result is appended with *history information*. This *history information* represents the current state of the state machine of the replica sending back the result. The user will eventually accept a result if one of the two conditions holds: 1. it receives $3f + 1$ (f is the number of faulty nodes) similar results with matching history information; 2. it receives between $2f + 1$ and $3f$ similar results with matching history information. The user sends a *commit* message to the replicas. Once $2f + 1$ replicas acknowledge receiving a *commit* message, the user accepts the result.

Since the requests may not be executed in the same order in all the replicas, the state of the correct replicas may diverge, and they may send different responses to the user. Therefore when a specific result is accepted, the history information appended to this result is forwarded to the replicas so they would update their current state to the one represented by the *history information*. This method reduces the number of messages used in BFT by 3.7 times and speeds up the time to achieve the correct result of a request to 2.7 times. The authors noted that their method is near optimal.

Unlike the previous work [19, 4, 5] focused on reducing the number of messages, Yin et al optimized the BFT [7] protocol by reducing replication costs. Their method [33] can tolerate f number of faults by only requiring $2f+1$ number of active replicas, while all the previous work presented require $3f+1$ active replicas. Their method separates the *ordering of the requests* from the *execution of the requests* in the BFT protocol. Separating the *ordering* from *execution* means that the nodes which cooperate in the BFT

protocol to order the requests (ordering nodes) are different from the nodes which actually execute the requests (replicas). Therefore, by separating these nodes, only *2f+1* replicas are needed to execute a request while still guaranteeing fault tolerance. In addition to reducing the number of replicas needed, their method solves the confidentiality problem in Amir's BLink protocol [5]. To achieve confidentiality, a privacy firewall is placed between the replicas and the ordering nodes. This firewall discards the results in minority and only forwards the result in majority to the user and the ordering nodes. This method prevents the ordering nodes and the user to find the faulty nodes.

In their protocol, a node (user) sends a request to the ordering nodes and they assign a sequence number to it. The assignment of the sequence number is achieved by the execution of the BFT protocol. When the order of the request is determined, it is forwarded to the replicas to be executed. When a replica receives a new request, it checks the request's sequence number *sn* with the sequence number *sn'* of the last request it calculated. If $sn = sn' + 1$ then the replica will calculate it; otherwise it will wait until it has calculated all the requests with a sequence number *sn''* that $sn < sn'' < sn'$. After calculating a result for the request, the node will send it to the privacy firewall. The firewall chooses the result in majority as the correct one and sends it to the user. Although separating the agreement from the execution reduces the number of replicas needed, it increases the time the user has to wait to receive the correct result of its request.

Similar to Yin [33], Correia also presented an efficient ordering protocol [8] which can mask *f* faulty nodes by using only *2f+1* replicas. Unlike Yin's [33] work, this protocol is not based on BFT but its performance is similar to Yin's protocol. To achieve the same ordering of requests in all the nodes, they make use of a *center node* which is responsible for the ordering of requests. In this protocol, a node (user) sends a request to one of the replicas. When the replica receives the request, it forwards it to all the other replicas.

When each replica has received the request, they hash the request and send the hash to the center. When the center receives *f+1* similar hashes it assigns the hashed request a sequence number and forwards it to the replicas. When a replica receives the sequence number of the request it executes the request in its given order. When a replica finishes the execution of the request, it sends the result to the user. Then the user waits for *f + 1* identical results from different replicas until it declares the result as correct.

As mentioned, although the efficiency of ordering protocols is not an issue for us, detecting faulty nodes and finding the correct result of each request is of utmost importance in both *stateless* and *stateful* fault tolerant systems. The techniques used for developing fault detection and finding correct results are similar in both *stateless* and *stateful* distributed systems.

The most common technique to detect faulty nodes is the use of *tests*. A test is designed to be capable of identifying faults in the nodes. Traditionally, a *central node* was responsible for conducting the tests and informing non-faulty nodes of the faulty ones which it had recently detected [9]. This central node is a single point of failure which is inconvenient for distributed systems. To adopt the traditional testing method with the current distributed computation systems, Hosseini et al presented a *distributed* fault detection method [15, 16]. In this method, any node can test a subset of other nodes for the presence of failures. The information used by a node to produce its fault diagnosis is the result produced by its *own* tests along with *information passed* to it indicating the results of the tests performed by other nodes.

In this protocol, a node *P* is tested by a group of nodes which is referred to as the testers of *P*. Each node only sends diagnosis information to its testers. When a node *Pj* sends a message to its tester *Pi* passing the information about the faultiness of a node (diagnostic message), *Pi* temporarily stores this message if *Pi* had found *Pj* to be fault-free when it last tested it. The next time when *Pi* tests *Pj*, if it finds *Pj* to be fault-free, it will consider

all messages received from *Pj* in the interval between the two tests as correct and will use them to produce new diagnosis information. Although this method prevents a single point of failure in fault detection, it works under the assumption that a node cannot repair itself before its faultiness is diagnosed by *another node*. This assumption is very strong and not realistic.

Albini et al optimized Hosseini's [15] distributed diagnosis method by reducing the number of tests. In their protocol, the nodes are grouped in clusters. One of the nodes is assigned as the *tester* of each cluster [10] and each cluster contains *N/2* nodes. Here *N* represents the total number of nodes. In each cluster, the tester tests all the nodes in the cluster and produces fault diagnosis information about each of them. At each testing interval, each tester of a cluster tests the testers of other clusters in addition to the nodes in its own cluster. When a tester *j* finds another tester *i* as fault-free, tester *j* asks tester *i* to send it the diagnostic information *i* has collected about its cluster. On the other hand, if the tester *i* is found faulty, tester *j* informs the nodes in the cluster to change their tester. When all the tests have been conducted, another protocol is run in the system which creates a graph whose vertices are the fault-free nodes and there is an edge directed from a vertex *a* to a vertex *b,* if *node a* has tested *node b* and has found it to be fault-free. Then the protocol finds the shortest paths between any vertex *i* and vertex *j* in the graph. The protocol keeps the edges in the shortest paths and deletes the rest. Then it propagates the diagnosis information of a node *i* to a node *j* through the nodes on *i*'s shortest path to *j*. This protocol runs until the diagnosis information of all the nodes has been propagated.

Since the testing method produces a great deal of additional network traffic, Kihlstrom et al presented a *new* family of fault detectors [17] which does not require the use of tests and is suitable for consensus problems [12]. In this method, they assume that each node has the capability of predicting the result for each request it sends out because it is familiar with the semantics of the algorithm which will calculate its request. In their fault

detectors, they consider three arrays in each node: one array representing all nodes that are faulty *(byz)*, one array representing all nodes that are suspicious of having crashed down *(outp)* and another array representing messages the node is expecting to receive *(exp)*. In the first step, when a node sends out a request to a node α, it turns on a timer. If the timer expires, the node appends α to *outp* and saves the details of the expected result in *exp*. When a node receives the result from α, it checks if the result meets its prediction. If it does, the node will accept the result; otherwise it adds the node to *byz*. Afterwards, it checks if the node is in the *outp* list. If so, it deletes it from the list and deletes the expecting result from α from the *exp* as well. In addition, it increases the timer for α for sending back a result. Every time one of the lists *outp* and *byz* gets updated, the node sets them to other nodes. When a node receives the *outp* and *byz* lists from other nodes, it adds the nodes in these lists to its own *outp* and *byz* lists. By passing around these two lists, all the nodes achieve a common view towards the system. Although their system presumably does not make use of tests, their assumption that each node can predict the correct result is very unrealistic. Therefore this method is not practical although it is more efficient than the testing technique.

Typical fault detection methods make use of excessive tests which only serve the purpose of finding the location and characteristics of a fault. These methods introduce a great deal of overhead into the system. Unlike previous techniques, we established a method which can eliminate the need for excessive test. It not only detects fault but also find the correct result to a request as well.

Finding the correct result of a request is a major issue in both stateless and stateful fault tolerant systems. The correct result to a request can be found in a centralized manner or a distributed manner. In this thesis, we use a centralized technique to find the correct result of requests. We will refer to the centralized technique as central decision making mechanism.

The most widely used centralized decision making mechanism is the *voting algorithm.* As mentioned, these algorithms mask incorrect results produced by faulty nodes, and prevent their propagation through out the system. These voting algorithms have been classified by Latif-Shabgahi [21] based on their functionality: generic voters, hybrid voters and purpose-built voters.

Generic voters [21] calculate the final result for a computation by only using the results received from nodes. These algorithms have fault detection capability and are also able to create warnings when they cannot make a decision. Note that in cases where a computation can have multiple correct results, these voters are inadequate. The best known of these voters is the majority voter. The *majority voter* chooses the correct result from among the variant input results, where at $\lceil (n+1)/2 \rceil$ inputs have the same value. In this thesis, we have adopted a majority voter as well. Because of its ability to detect faults, we chose this voter to integrate fault detection and finding the correct result.

Another well known generic voter is the *median voter.* This voter selects the mid-value of the variant input results as the final result. This voter can produce the correct result when a maximum $\lfloor (n+1)/2 \rfloor$ input results are incorrect. Although this voter can handle cases where a computation can have multiple correct results, it does not have fault detection capability. In addition this voter has a higher probability to produce incorrect results compared to the majority voter.

Hybrid voters [21] use additional information such as the reliability level of nodes, on-line diagnosis information of the nodes, or other various probabilistic information to improve their performance. These voters optimize genetic voters and produce more accurate results compared to them. The best known voter is the *weighted average voter* which calculates the weighted average of the input results. The weights can be predetermined or can be dynamically assigned. Typically, these weights represent the possibility of the input result being faulty. A well known voting algorithm which uses weights is the *adoptive majority voter* introduced by Latif-Shabgahi [22]. This adaptive majority voter uses the node's history of faulty behavior to improve the performance of

the majority voter. In their method, the number of times a node has produced the correct result is computed. The nodes with the most produced correct results are considered as most reliable and the nodes with the least produced correct results are considered as the least reliable. In their method, a majority voter with a predefined voting threshold α is used. If the majority of n nodes produce results with a difference of α of each other, based on a majority voter, all of the n nodes are producing the correct result, and the voter should choose one of these results as the correct one. Previously, majority voters randomly choose the final result from the set of correct results, but by considering the reliability of the nodes, this voter can be improved. Instead of randomly choosing the final output, the voter can choose the result of the node with the highest reliability as the correct result.

Voting algorithms incorporating prediction and smoothing [21] are another group of hybrid voters. These voters are suitable for cyclic systems in which there exists some relationship between the result in one cycle and the result in the next. Knowledge of this relationship between successive results is used in this class of voters. A well known voter in this group is the *smoothing voter.* This voter extends the majority voter by adding an acceptance test. This test indicates a node as faulty if the node executes the same request twice and there is an excessive difference between the two results it produces. These voters are used in systems where one computation is executed several times by the nodes. When a decision cannot be made by simple majority voting, the smoothing voter chooses the input result closest to the previous output produced by the voter as the final result for the current computation. We do not make use of the voters in this category because we have assumed that all the computations in the system are non-cyclic. Since our method is independent of the type of computations (cyclic or non-cyclic), our method can be simply adapted to systems with cyclic computations.

Another well known algorithm in the hybrid voter category is the *integrated voting algorithm*. These voters use self-diagnostic methods which enables them to select correct results more often compared to genetic voters. Latif-Shabgahi et al presented a family of

these voters [22]. It is not important how the diagnosis information is created; the important factor to know about this information is that they present the percentage of fault in each node. The authors present a limit for diagnosis information: any node with diagnosis information higher than this limit is considered reliable. They present four ways on how this diagnosis information can be used to optimize voting performance:

The diagnosis information of nodes is handled independently from the results returned. In this case, the voter finds the final output by only using the input results. Then the voter uses the diagnosis information to produce warning in the case when the diagnosis information show severe damages in the node whose result is chosen as the final output.

The voter chooses the output result based on the input results and the diagnosis information of the node. The diagnosis information is used to validate the output result. This method can work in two ways: (i) the voter finds the output result and if among the nodes which have given the correct result, there is a node whose diagnosis information is higher than the limit, the final result is validated; otherwise the voter produces a warning. (ii) the second way is similar to the previous one. The difference is that in the case where none of the nodes with the correct result has a diagnosis information higher than the limit, the result returned by a node with the highest diagnosis information is validated and chosen as the output.

The node with the highest diagnosis information is found and its result is chosen as the output result. In this case, the voter is nothing more than a selection function.

The voter uses the diagnosis information to improve the method it uses to find the output result. These voters use the diagnosis information of each node to weigh its result returned. The higher the diagnosis information is, the more weight the result has.

In some cases, designing a voter specialized for a specific application domain can result in more optimal results. These voters are referred to as *Purpose-built voters* [21]. These voters are customized based on the characteristics of the application. Some sample of voters in this category can be: *SIFT* [21]*:* It is a voter in aircraft control which is a

customized majority voter. *Stepwise negotiating voter* [21]*:* It integrates the majority voter and standby redundant systems. A *standby system* halts a number of its components when the voter can not make a decision. When the voter finds a result, the halted components will continue functioning again. The Stepwise voter behaves as if it were a majority voter if it can find the output result; but if it cannot, the system degrades to a standby system. The *expedient voter* [21] is used in multi-stage software systems. In these systems, each software is not considered as one unit; instead it is subdivided into components and the voter produces an output once enough number of components has produced a result. It has been seen that this voter works considerably faster than the case where the software is considered as one unit.

Although a voter is a less complicated method to find a correct result compared to a distributed architecture, a distributed architecture is more compatible with distributed computation systems.

Finding the correct result to a request in a distributed manner can be generalized to the *byzantine agreement problem*. The *byzantine agreement protocol* tries to arrange all the non-faulty nodes to agree on the value of a certain entity [20]. For example, the correctness of a node (is it faulty of not), the result of a request, the order of the requests, etc. Lamport's protocol is the most well known byzantine agreement protocol [20]. It is *round* optimal in the sense that the number of rounds the protocol runs until an *agreement* is reached is optimal. By *agreement* we mean that all the nodes accept a common result as the correct one of a request. In their method, when a server calculates a request, it signs the result and sends the result to other nodes. When a node receives a result from the server it logs it and then signs the result and forwards it to all the other nodes. When the node receives a signed result which is not equal to the result it originally received from the server, it adds the newly received result to its log. If the signed result has not been signed by all the nodes yet, the node will sign it and forward it to others. Otherwise, if the result is signed by everyone, the node checks to see how many different results it

has logged. If more than one result is signed by the server, it can define the server as faulty and discard all the results. Otherwise, in the case that there is only one result signed by the server, the node chooses that as the correct result of its request. This protocol can tolerate faults up to one third of the nodes. Their method has exponential message complexity therefore making it impractical.

One of the early and well known work presented in optimizing Lamport's [20] protocol is Lynch et al [24]. They were capable of changing Lamport's protocol in a way that its message complexity were polynomial. In the first step, when a node receives a result from the server, it signs the result and forwards it to the other nodes. When a node receives a signed result for another node, it saves the result in its log. When the number of matching signed results exceed $f+1$, the node signs the result and sends it to the other nodes. Here f is the number of faulty nodes. When a node receives $2f+1$ number of signed results from distinct nodes, it will send out a *commit* message indicating that it has accepted the result. When a node receives $2f+1$ *commit* messages, it will then use the result. Unlike Lamport's protocol, this methods is practical since it has a polynomial message complexity.

In addition to Lynch's [24] work, Fitzi also introduced a byzantine agreement protocol with a polynomial message complexity in terms of the number of nodes [13]. In their method, the protocol has two phases: information gathering and data convergence. Each node has a tree where each vertex represents the result returned by a node. The root vertex represents the result returned by the server and each vertex shows a result returned by a node. The result in a vertex for a node denotes that this is the result that the server returned to the node. For instance, a vertex $i.j.\alpha$, denotes that a node i says that node j says that the server returned the value α to it.

The information gathering phase has multiple rounds. In the first round, the server sends its result to all the nodes and they save it in the root vertex. In the next rounds, the nodes

send their trees to each other. Each node will update its tree using the trees it has received. After each round of information gathering, in the data convergence stage, if the difference between the data received by another node and the result in the root node is smaller than σ then the result is untouched; otherwise it is set to $\perp$ which is a value out of the possible result range. Vertices with the $\perp$ value are noted as faulty. The updated tree is sent out in the next round of information gathering. In this step, to reduce their message complexity to polynomial, they prune their tree by deleting vertices from nodes which seem to have crashed or are faulty. Therefore messages are only sent to nodes assumed to be fault-free. The execution of these two phases continues for a predefined number of times. Although this method runs in polynomial time, it is not an optimal solution.

Feldman presented an *optimal* byzantine agreement method [11] which can reach agreement in synchronous systems if the number of faulty nodes is smaller than one third of the total number of nodes; and if the system is asynchronous, agreement can be achieved if the number of faulty nodes is smaller than one fourth of the total number of nodes. Their protocol guarantees that: 1. if a value is produced by a non-faulty node, other non-faulty nodes will agree on the value; 2. if a value is produced by a faulty node and a non-faulty node agrees on the value, other non-faulty nodes may or may not agree on it. In the first step, when a node produces a value, it sends the value to the other nodes. In the next step, all the nodes share their received values with each other. Then each node is given a random number from 0 to n (total number of nodes). The node with the least number chooses *n-f* values from those other nodes have shared with. Here *f* is the number of faulty nodes. This node chooses values close to the one it originally received and sends its chosen values to other nodes. When the nodes receive the chosen values, they calculate the average of the chosen values. If the average has a difference smaller than α from the original value they received, they will consider the average as the correct result for the request. Otherwise the node will discard the received values. In this case, although the proposed protocol can achieve agreement in constant time with optimal message and

cryptography complexity, in *1-t/n* of the times, the agreement cannot be achieved.

Hardekopf et al presented a *new* family of byzantine consensus protocols [14]. His protocol is not based on the traditional Lampton protocol [20] and can reach agreement in constant time just like Feldman's [11] protocol. In this work, the author combines byzantine agreement with a central decision making mechanism. Every node calculates the result for a computation request independently and sends its results to other nodes. Every time a node receives a result from another node, it runs its central decision making mechanism (typically a voter) to see if it can find the correct result. When the central mechanism finds the result, the node sends it to a shared memory and a timer in the memory starts to count down. If a result resides in the memory, it is over-written by the new result. In the next steps, when a central mechanism of a node finds a result, it checks its result with the one in the shared memory. If its result is equal to the one in the shared memory, the node does nothing; otherwise it replaces the old result in the shared memory with its own and forwards its result to other nodes based on Lynch's byzantine agreement protocol. When the timer of the memory reaches zero, the result in the memory is sent to the user. The authors have shown that their algorithm has an average O(1) complexity in relation with the number of nodes.

Hardekopf further on advances their method to avoid the use of shared memory [14]. In the new method, each node calculates a result for a computation request and saves the hash of its result. In the next step, the node sends its result to other nodes based on the byzantine agreement protocol. When a node receives a result, it checks it with its own result, if they are similar, the node creates a hash of the received result and sends it back to the sender. This hash is called an endorsement. It shows that a node has calculated the same result. When a node receives an endorsement, it runs its central mechanism (voter) to see if it can find the correct result based on the received endorsements. If the voter can find a result, the node forwards it to the node which requested the computation. The authors show that the algorithm complexity in each node is O(n), where n is the total

number of nodes.

In addition to the above pieces of work, Sobe also aimed at optimizing the process of finding the correct result by reducing the number of messages passed around [31]. Sobe reduced the message complexity by eliminating the messages containing information about the correctness of a result. These messages are piggybacked on the messages containing requests and results. Although their method reduces the number of messages, it introduces new problems: a node must wait for the next message to arrive until it can find out that a previous result it has received is correct or not. This message can take a great deal of time to arrive, therefore introducing unnecessary wait. Sobe tried to solve this problem by allowing the nodes to make use of the result. If later on the result is found incorrect, the node must rollback all the calculations it has done based on the result. This is not a very convenient solution because an incorrect result can be propagated to other nodes. Therefore those nodes must be notified and must rollback as well. In this situation the effect of an incorrect result can be similar to a ripple effect. Therefore repairing the system can be very costly and time consuming.

All the previous work presented have been designed and evaluated in traditional distributed computation systems. Today, a great deal of distributed computing is being carried out by web services. These services are incapable of handling the presented fault tolerant mechanisms because of the heavy computations embedded in them. Therefore new fault tolerant mechanisms need to be presented for web services.

Zhao presented a fault tolerant method for stateful web services [34] based on the BFT [7] protocol presented by Castro [7]. This method adopts BFT to the web service architecture by reducing the number of message passing and message processing. As in BFT, a service request can only be executed when the three phases (pre-prepare, prepare, commit) of the BFT protocol has been completed. But unlike in BFT where all the

replicas which execute the request send back the result, in this method a replica only logs the result and does not send it to the client until the replica becomes a primary after a view change. If the primary becomes faulty, the user will accept an incorrect result since only the primary sends back the results. To solve this problem, if the client uses a central decision mechanism (such as voter), the client must wait for $f+1$ (f is the number of faulty nodes) view changes until it can find the correct result for a request. This number of view change may take a great amount of time.

Merideth et al [24] pointed out that traditional fault tolerant methods are not suitable for web service architecture because stateful web services are typically multi-tier, in the sense that a web service can be dependent on other web services, therefore creating a multi-tier infrastructure [24]. They show that Zhao's [34] method only supports fault tolerance in the first tier. Therefore a method is needed to advance fault tolerance to the remaining tiers as well. In their architecture a web service is replicated on levels one and two but web services on the third tier and above is provided by only one node. In their method a node (user) sends its request to the replicated web services using the BFT [7] protocol. In the next step each replica sends a request to each web service β which it is dependent on. Each web service β executes the service after receiving $f+1$ requests for it and then sends back the result to the requesting replicas. When the requesting replicas receive the result, each replica checks its result from β with the other replicas. If it matches the majority of the results, the replica uses the result from β to compute the final output for the user's request. Similar to BFT, the user waits for $f+1$ similar results before accepting it as the correct one. In this method, since the web services on the third tier and above are provided by one node, it can suffer from unavailability if a node in these tiers crash.

In addition, they present spatial and temporal redundancy techniques for establishing stateless fault tolerant web services [24]. In spatial redundancy, they use fault masking techniques to prevent the propagation of faulty results. While web services are replicated

on different machines, only one web service (called the primary web service) provides the requested service at a time. A redundancy manager is considered to check the correctness of the primary by repeatedly testing it. If a service shows to be faulty, the redundancy manager discards the result from the web service and allocates another web service as the primary and resends the request to the new primary.

In temporal redundancy, a second attempt at a request is repeatedly initiated until the correct result is obtained. With this technique, a service has no physical replicas. At first, a node (user) sends a request to the web service and waits for the result. When the user receives the result, it tests the web service. If it shows to be faulty, it will resend the request to the web service again and again until the web service shows to be fault-free. Since each web service is stateless, the only way to make such a service fault tolerant is to retry to use it until it gives the correct result.

As for stateless web services, Santos et al presented a fault tolerant method which is independent of a protocol which established fault tolerance [28]. The authors argued that methods such as Zhao's [34] and Meredith's [24] are against the methodology of web services: they should be interoperable and independent of various technologies and platforms. Their newly proposed method uses technologies such as XML, SOAP, UDDI, etc used by all web services. In their method, they considered a central node which is responsible for invoking service replicas, analyzing the results for each request and detecting faults. In the first step, a node (user) sends its request to the central node which forwards the request to all the replicas. When a replica finishes the execution of the request, it sends back the results to the central node so it will use its voter to find the correct one. Since the voter does not play the role of a fault detector, the central node has other components to detect faults. These components test the replicas periodically to check if they are faulty or not and take the faulty replicas out of the system. Their method suffers from the excessive network traffic caused by tests carried out by the fault detection components. The periodic test not only causes excessive messages but also

greatly delays the time to reach a result. Our method uses a similar technique to find the correct result of a request but avoids additional message passing for fault detection by incorporating fault detection capabilities into our voter.

Our method is capable of tolerating *f* faults with only *2f+1* replicas, similar to Yin's [33] and Correia's [8] protocols. Since our method is designed for stateless fault tolerant services, it is not based on the BFT [7] protocol: we are not concerned about the ordering of the requests. As a result, our method only makes use of replicas and not ordering nodes [7]. Consequently, we only need *2f+1* replicas for executing the requests, resulting in a more efficient protocol compared to BFT [7] and Amir's [4, 5] protocol in terms of the number of active replicas needed.

Our method integrates fault detection techniques with the techniques to find the correct result of a request. Our fault detection method is similar to Hosseini's [14, 15]. The major difference is that each node diagnoses itself as faulty and does not rely on other nodes to find its faultiness. Therefore we do not have to make use of their strong assumption that no node can repair before being diagnosed by other nodes. The method we use to find the correct result is similar to Santos [28], but our central node does not have separate components to test for faulty nodes. The voter in the central node is a genetic voter (the majority voter). At this stage a simple genetic voter was enough for us to achieve our goals. In our method, a voter can find the correct result in an amount of time less than that by the genetic voters, and thus increases the performance. To our understanding, all the voters presented have the same approximate speed when having *f* faulty nodes in the system. But unlike Santos [28], the voter in our central node can not only find the correct result, but also detect faulty nodes. With the use of this method, the messages sent out in the system are used both for finding the correct result and detecting faulty nodes, therefore eliminating the need for additional messages to detect faults. The message complexity of our method is equal to the state-of-the-art efficient fault detection methods presented. We have a message complexity of $O(n^2)$ where *n* is the number of nodes while

providing a distributed fault detection technique.

Chapter 3. Problem Definition and Our Solution

Although fault tolerant mechanism can help diagnose and mask faults to enable critical systems to function without explicitly showing errors, they normally demand for a great deal of extra computational time and additional system resources. The overhead that a fault tolerant mechanism introduces can make a system very costly to operate. In particular, a fault tolerant system can become inappropriate for highly interactive and time sensitive applications because of the additional time it takes to achieve fault tolerance. Therefore, there has been much concern about reducing the overhead of the fault tolerant mechanisms.

There are several ways that fault tolerant mechanism can introduce overhead:

- requiring additional *resources and computation power to calculate the result* of a request. As mentioned before, the basic technique to establish fault tolerance is *redundant computation* which requires that a request be computed in multiple nodes. Therefore it is clear that more resources are needed to establish a fault tolerant system;
- requiring additional *computation power for the voting and test method*. These methods diagnose and mask faulty results by analyzing the data, taking into account the identities of the nodes, the results they have returned, etc. Additional computation power is consumed to carry out such analyses.
- requiring additional *time* to mask faulty results and find the final result for a computation request. In regular (i.e. non-fault-tolerant) systems, a request is computed by only one node and its result is used by the user at this node. In fault tolerant systems, on the other hand, a request is computed by multiple nodes and a single result is derived using the results computed by multiple nodes. Therefore, considering that different nodes have different computational speeds, from slow PCs to high speed frameworks, the time to derive the final result increases, as the user should wait for more than one node to calculate the request. In addition, after the results are acquired, a certain amount of time is consumed to diagnose the faulty results, and choose one as the correct result for the request.

As it can be seen, by considering such additional computation time, the time to reach a result for a request is largely increased compared to that for regular systems.

- *sending additional messages* over the network. To achieve fault tolerance, additional information about the identities of the nodes, the requests, the related results etc. need to be passed among different nodes.
    - As the system uses redundant computation, a request is sent to multiple nodes instead of only one, introducing new messages.
    - Consequently, each node has to send back its result to the voter. This introduces a group of additional messages.
    - When the system uses weighted voters, the voters make use of the information about the nodes, such as their fault percentage, their probability of becoming faulty, etc. For such information to be sent to the voter, another group of messages are introduced into the system.

As it can be clearly seen, to reach fault tolerance in the system, lots of messages have to be introduced.

Critical systems in need of fault tolerance are usually *time sensitive systems*, such as space crafts and nuclear reactors, where the time to get a result is vital for the system to properly function. Therefore, it is highly desirable that these systems do not spend too much additional time for applying a fault tolerant method to find the correct result of a request: It is quite important that the amount of time to mask faults is well-controlled so that the correct result can be reached in time. For instance, consider a space craft in which its sensors detect a solar storm. The fault tolerant method takes a considerable amount of time to diagnose the incorrect sensor data and chose the final result sent back from the sensors indicating that a solar storm is coming. Therefore the space craft cannot take appropriate actions in time and it is dislocated from the earth's orbit by the solar storm.

Anther commonly known characteristics of critical systems is carrying out *intensive computation*. For example, consider a computational base station which analyzes the data

sent back from a space satellite. In a regular mode, this station makes use of two high speed computational frameworks to analyze the data. To achieve fault tolerance, suppose that the data need to be analyzed by ten high speed computer frameworks and their results are checked by a fault tolerant method in another high speed framework to find faulty results and choose a final one. This fault tolerant method requires 11 frameworks to analyze each data sent back from the satellite, which is a considerable amount of computation overhead. As such systems make use of very expensive computing resources, avoiding any additional computations is highly desired.

Although the overhead introduced by fault tolerance cannot be completely avoided, it can be reduced by using well-studied methods that take into account the time complexity and resource (network bandwidth, memory space, computational power, etc) use.

This thesis focuses on developing an *efficient* fault tolerant method. Our goals are as follows:

1. *reducing the time to reach a result for a request:* we consider reducing the time to reach a correct result by reducing the time to diagnose and mask faults.
2. *reducing the consumption of computational power to calculate requests:* we consider eliminating those computations that do not affect the decision of the final result for that request.

We assume that the system considered has a *central* voter in the sense that it collects the results calculated by every node and chooses a final result using quorum-based voting mechanism.

Different nodes can provide different services. For example, node A can process requests q_1 and q_2 while node B can process requests q_1, q_3, and q_4.

We assume that each node can process a *predefined* set of requests and we assume that all nodes have the knowledge about which other nodes are capable of processing which requests.

Each node can issue a service request to *m* nodes that can process it, where *m* is at least twice the number of nodes that may become faulty at the same time so that the majority voting can be carried out.

A request can be initiated at any node. Multiple requests can be initiated at different nodes at more or less the same time. Due to network delay, a same request can be received by different nodes in different orders.

When a central voter found a correct result of a computation, it sends the result to every node in the system.

(Ass1) When a node starts to malfunction, it will never produce a correct result until it is repaired.

(Ass2) A malfunctioning node can only be repaired by the administrator.

Our solution to provide reduction on the consumption of both time and computation power are based on two observations.

> *Observation 1*: If a result calculated by a node at time *t* turns out to be correct, then for any time t_1 before *t*, all the results calculated by this node are correct as long as no reparation was carried out between t_1 and *t*.

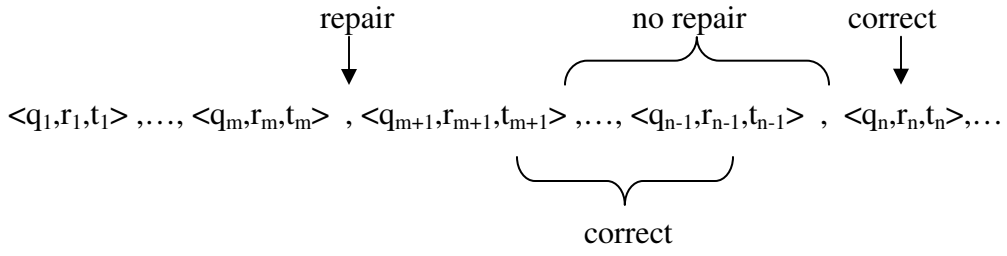

Figure 1 Observation 1

This is illustrated in Figure 1: if r_n is correct then r_j is correct for all j such that $m+1 < j < n-1$.

Let <q,r,t> denote the computation of request q at time t, resulting in r. Consider the situation where node N has calculated the following results: $<q_1,r_1, t_1>$, $<q_2,r_2, t_2>$, $<q_3,r_3, t_3>$, $<q_4,r_4, t_4>$, $<q_5,r_5, t_5>$ where $t_1< t_2< t_3< t_4< t_5$. After time t_5, we realized that the result r_4 for q_4 is correct. Suppose that N has only been repaired at a time between t_2 and t_3: no reparation is done between time t_3 and t_4. Then based on the above observation, we can conclude that the result r_3 is correct for request q_3.

The correctness of observation 1 is derived from assumptions *Ass1* and *Ass2*: When a node realizes that a result *r* it has calculated is correct, based on assumption *Ass1,* we can conclude that this node was functioning properly at the time it calculated *r.* Now based on assumption *(Ass2),* it can be concluded that the node was functioning properly between the last reparation and the time it calculated *r*. Therefore, based on *Ass1,* we can conclude that all the results calculated in this time period are correct.

> *Observation 2*: If a result calculated by a node at time *t* turns out to be incorrect, then for any time t1 after *t*, all the results calculated by this node are incorrect as long as no reparation was carried out between *t* and *t1*.

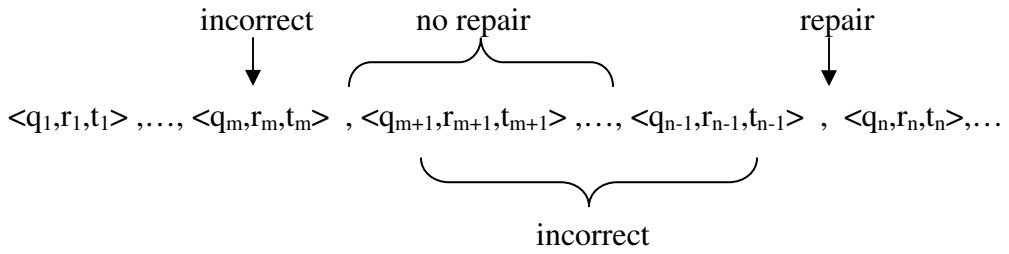

Figure 2 Observation 2

For instance, consider the previous example. Suppose now that node N realizes that its result r_3 for q_3 is incorrect and that it has only been repaired at a time between t_4 and t_5. Based on this observation, it can be concluded that the result r_4 is incorrect for request q_4.

The correctness of observation 2 is also derived from assumptions *Ass1* and *Ass2*: When a node realizes that a result *r* it has calculated is incorrect, based on *Ass1,* we know that the node was malfunctioning at the time *t* it calculated *r*. According to *Ass2,* we can conclude that the node is malfunctioning from time *t* until the time the node was repaired. Using *Ass1* again, we know that all results calculated from time *t* till the time the node was repaired, are incorrect.

For convenience, we will use α, β to range over services, ρ to range over nodes. We will use

- $T(\rho, t)$ to represent the type of the fault at node ρ at time t. In particular, we use a special symbol Null to denote that there is no fault at node ρ.
- $R(\rho, t, \alpha)$ to represent the result of the computation of α at node ρ at time t.
- $C(\rho, \alpha)$ to represent the correct result of computing α at node ρ.
- $fix(\alpha, t)$ to represent whether node α is fixed at time t (and will function well afterwards).

With these notations, the two observations are formally expressed as follows.

Proposition 1:

$\forall t, t_1,$ *if* $R(\rho, t, \alpha) = C(\rho, \alpha)$

and $\forall t_2$ *s.t.* $t_1 < t_2 < t$. $fix(\alpha, t_2) = false$

then $\forall \beta, \forall t_3$ *s.t.* $t_1 < t_3 < t$. $R(\rho, t_3, \beta) = C(\rho, \beta)$

Proposition 2:

$\forall t, t_1,$ *if* $R(\rho, t, \alpha) \neq C(\rho, \alpha)$

and $\forall t_2$ *s.t.* $t < t_2 < t_1$. $fix(\alpha, t_2) = false$

then $\forall \beta, \forall t_3$ *s.t.* $t < t_3 < t_1$. $R(\rho, t_3, \beta) \neq C(\rho, \beta)$

The following example highlights how these two propositions can be used in fault tolerant distributed systems to improve the efficiency.

Example 1: Assume that the network has six nodes N1, N2, N3, N4, N5, N6 and there are eight requests q_1, q_2, q_3, q_4, q_5, q_6, q_7, q_8 handled in a certain period. Suppose that the fault tolerant distributed system goes through the following steps:

We use tuple $<q_1, r_{1,i}, t, b>$ (called message-tuple below) to denote that result r_{1i} has been calculated for request q_1 at time t. Here t has values $t_1, t_2, \ldots t_n$ where $t_1<t_2<t_3\ldots.<t_n$. In addition, b has three values: when b is null, it means we do not know yet whether r_{1i} is correct or not. When b is true or false, it means that r_{1i} is correct or incorrect, respectively.

Step1: each node sends out its requests to other nodes. In table 2, the row of Step 1 shows the requests sent by each node.
Step 2: each node receives the requests. In table 2, the row of Step 2 shows the requests received in order by each node. This is also illustrated in Figure 3 where the arrows illustrate the requests sent to and received from node N1.

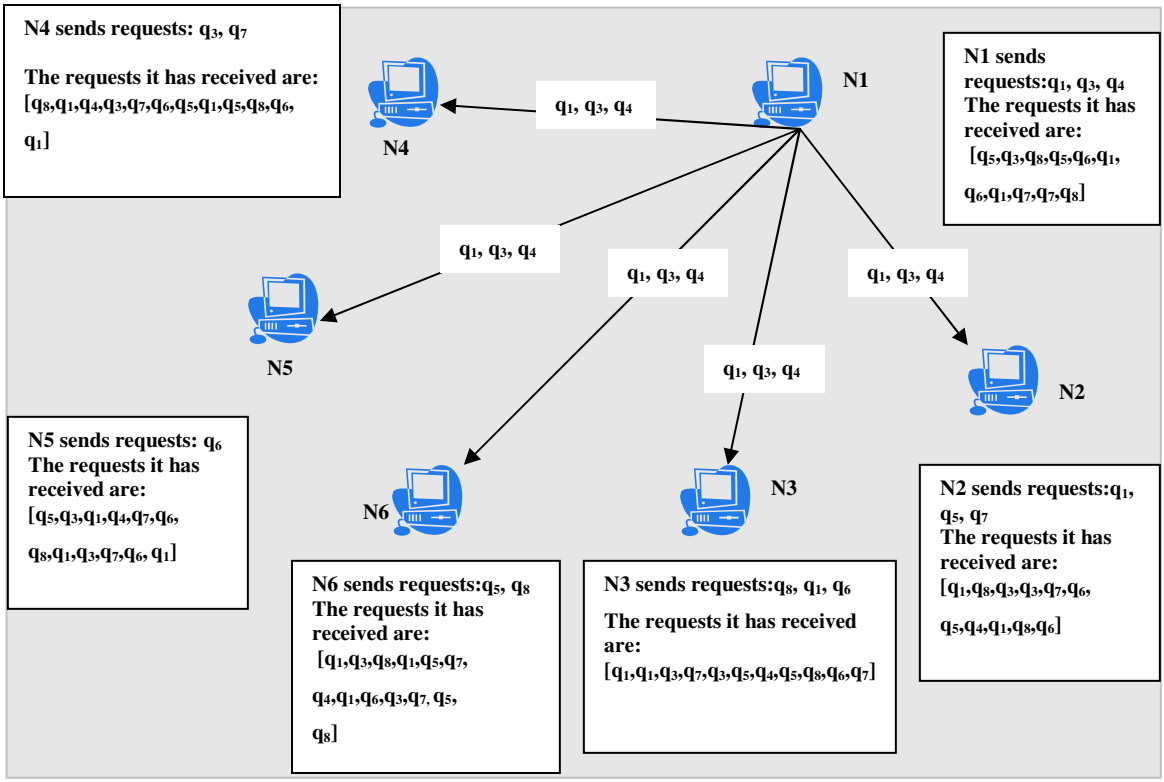

Figure 3 Requests sent in the system

Step 3: each node, in a certain period of time, finishes the computation of some requests, and sends the results to the voting center. In table 2, the row of Step 3 shows the message-tuples calculated for the first several requests received. This step is illustrated in the following picture.

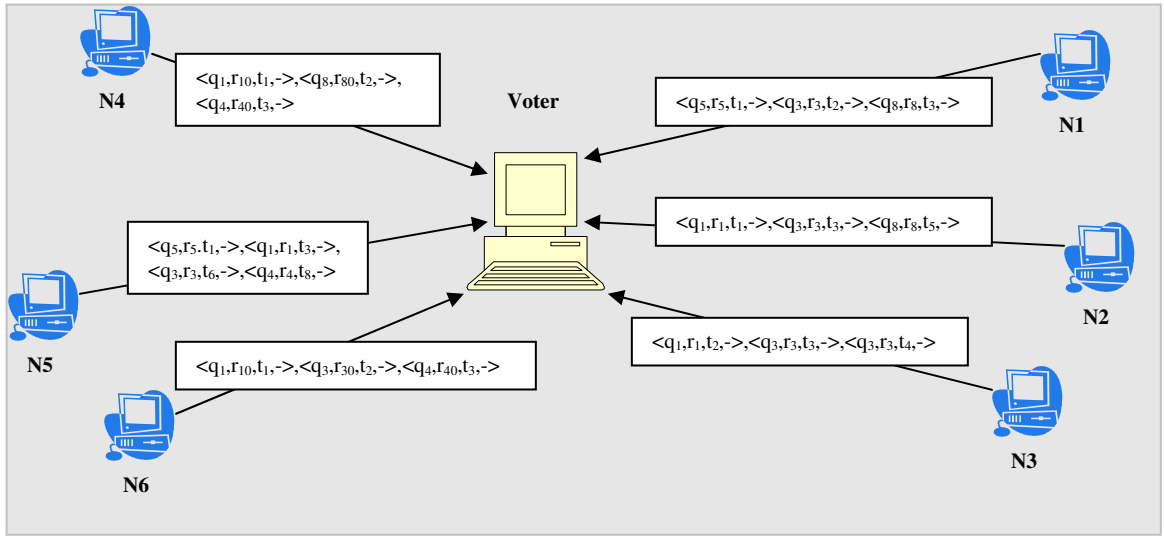

Figure 4 Results sent to the voter

Step4: the voting center receives the results of Step 3 from all nodes. In table 2, the row of Step 4 shows these results received by the voting center.
Step 5: the voting center finds the correct results for some of the requests using its voting algorithm. In table 2, the row of Step 5 shows the message-tuples consisting of the correct results that the voter is able to derive at this time. These results are sent back to all other nodes.

Table 1 shows the decisions of the correct results made by the central voter at this stage. There are 6 nodes in total so in order for a result to be considered correct, it has to be returned by at least three nodes. For example, the majority voter finds four nodes having calculated r_1 for q_1. Therefore it chooses r_1 as the correct result for q_1.

Requests	Results Received	Correct Result	Messages sent from voter
q_1	r_1, r_1, r_1, r_1 r_{10}, r_{10}	r_1	<q_1,r_1,-,true>
q_3	r_3,r_3,r_3,r_3,r_3 r_{30}	r_3	<q_3,r_3,-,true>
q_4	r_4 r_{40}, r_{40}	n/a	No message sent
q_5	r_5,r_5	n/a	No message sent
q_8	r_8,r_8 r_{80}	n/a	No message sent

Table 1

Picture 5 illustrates the messages sent by the voter after having calculated the correct results.

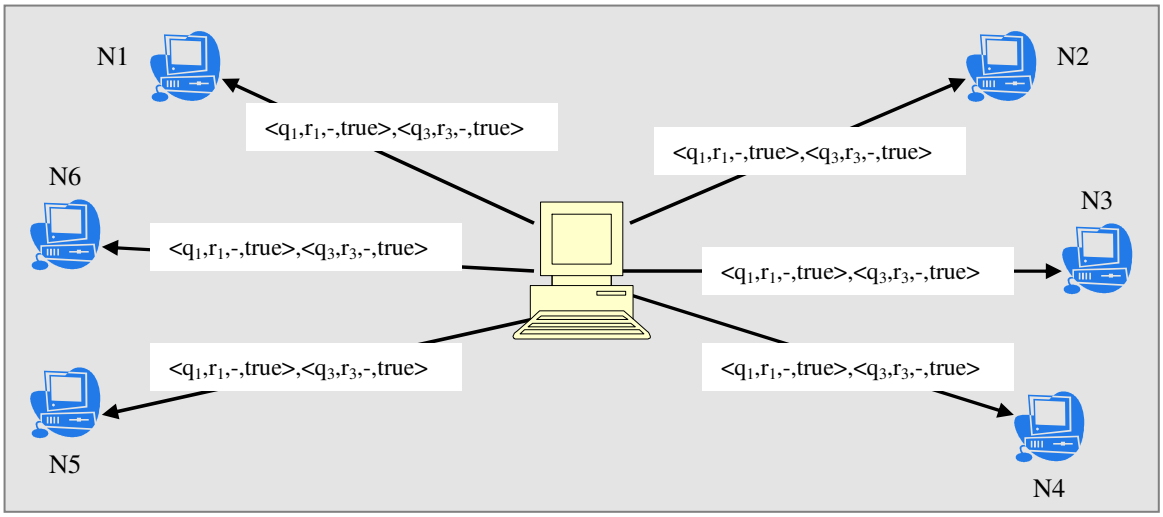

Figure 5 Results sent by the voter

Step 6: each node derives the correct/incorrect results using Proposition 1 and Proposition 2, and sends the results to the voting center, if there are any. In table 2, the row of Step 6 shows the message-tuples consisting of the correct/incorrect results found by each node at this time.

Having received correct results for q_1 and q_3, nodes N1, N2 and N5 are able to conclude, by using Proposition 1, that their results for requests q_5 and q_8 are correct. Nodes N4 and N6 can conclude, by using Proposition 2, that their result r_{40} for request q_4 is incorrect. The following picture illustrates how each node uses Proposition 1 and Proposition 2 to derive correct/incorrect results. Here a red circle highlights a request in a node whose result is judged by the voter as correct. A blue circle highlights a request whose result is judged by the voter as incorrect. In the end of this step, two more results are found correct: $<q_5,r_5,t_1,true>$ and $<q_8,r_8,t_3,true>$.

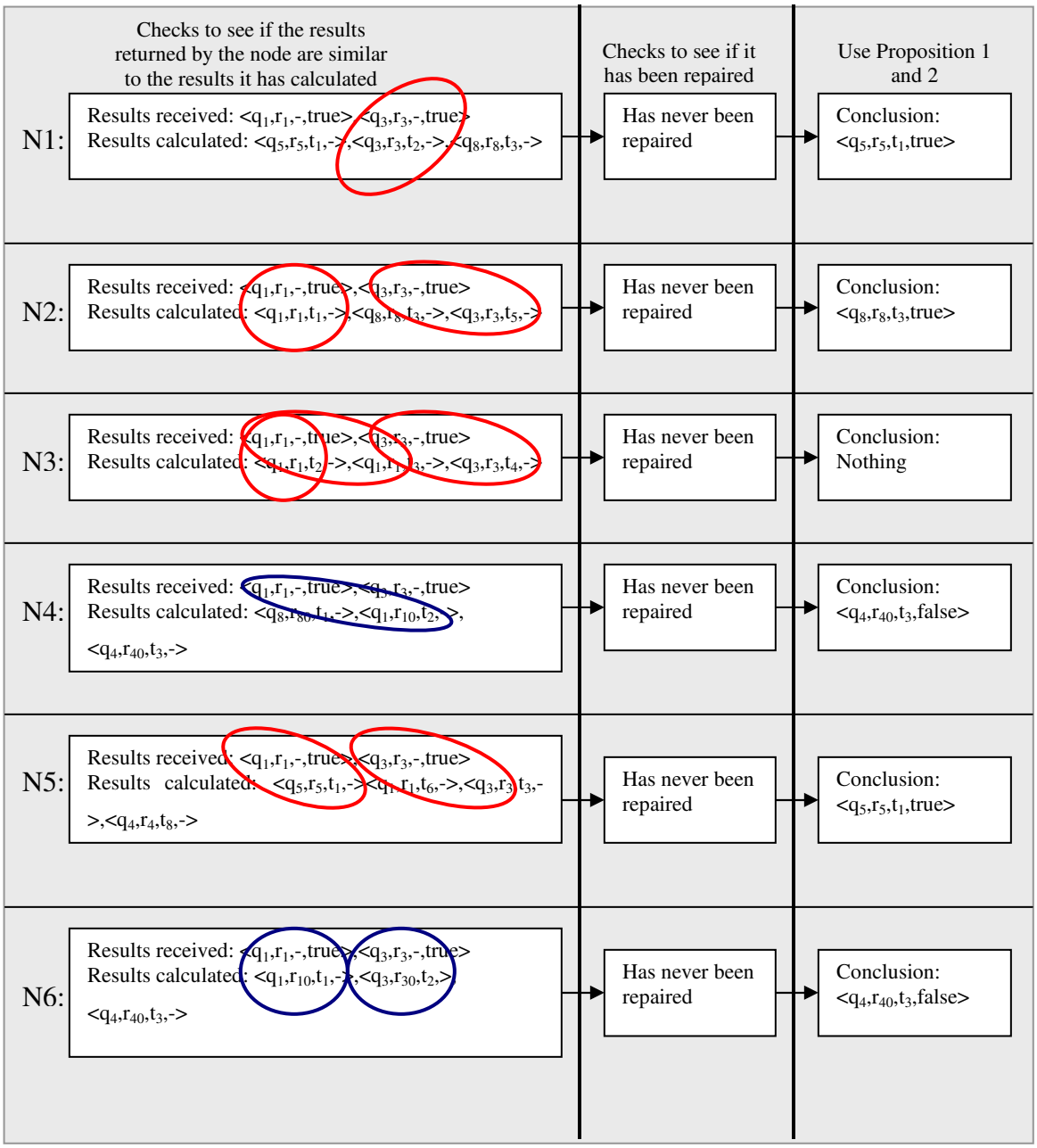

Figure 6 Using Proposition 1 and 2

Step 7: the voting center sends the correct results $<q_5,r_5,t_1,true>$ and $<q_8,r_8,t_3,true>$ achieved in Step 6 to all the nodes. In table 2, the row of Step 7 shows the message-tuples with all correct results found in Step 6.

Steps	N1	N2	N3	N4	N5	N6	Voting Center

1	q_1, q_3, q_4	q_1, q_5, q_7	q_8, q_1, q_6	q_3, q_7,	q_5, q_8	q_6	-
2	[q_5,q_3,q_8, q_5,q_6,q_1, q_6,q_1,q_7, q_7,q_8]	[q_1,q_8,q_3, q_3,q_7,q_6, q_5,q_4,q_1, q_8,q_6]	[q_1,q_1,q_3, q_7,q_3,q_5, q_4,q_5,q_8, q_6,q_7]	[q_8,q_1,q_4, q_3,q_7,q_6, q_5,q_1,q_5, q_8,q_6, q_1]	[q_5,q_3,q_1, q_4,q_7,q_6, q_8,q_1,q_3, q_7,q_6, q_1]	[q_1,q_3,q_8, q_1,q_5,q_7, q_4,q_1,q_6, q_3,q_7, q_5, q_8]	-
3	<q_5,r_5,t_1,->, <q_3,r_3,t_2,->, <q_8,r_8,t_3,->	<q_1,r_1,t_1,->, <q_8,r_8,t_3,->, <q_3,r_3,t_5,->	<q_1,r_1,t_2,->, <q_1,r_1,t_3,->, <q_3,r_3,t_4,->	<q_8,r_{80},t_1,->, <q_1,r_{10},t_2,->, <q_4,r_{40},t_3,->	<q_5,r_5,t_1,->, <q_3,r_3,t_3,->, <q_1,r_1,t_6,-> <q_4,r_4,t_8,->	<q_1,r_{10},t_1,->, <q_3,r_{30},t_2,->, <q_4,r_{40},t_3,->	
4	-	-	-	-	-	-	<q_5,r_5,t_1,->, <q_1,r_1,t_1,->, <q_8,r_{80},t_1,->, <q_1,r_1,t_2,->, <q_1,r_{10},t_1,->, <q_5,r_5,t_1,->, <q_3,r_3,t_2,->, <q_8,r_8,t_3,->, <q_1,r_1,t_3,->, <q_3,r_{30},t_2,->, <q_1,r_{10},t_2,->, <q_3,r_3,t_3,->, <q_3,r_3,t_4,->, <q_8,r_8,t_3,->, <q_3,r_3,t_5,-> <q_4,r_{40},t_3,->, <q_4,r_{40},t_3,->, <q_4,r_4,t_8,->, <q_1,r_1,t_6,->,
5	-	-	-	-	-	-	<q_1,r_1,-,true>, <q_3,r_3,-,true>
6	<q_5,r_5,t_1,true>	<q_8,r_8,t_3,true>	-	<q_4,r_{40},t_3,false>	<q_5,r_5,t_1,true>	<q_4,r_{40},t_3, false>	-
7							<q_5,r_5,-,true>, <q_8,r_8,-,true>

Table 2

Chapter 4. Implementation

In this chapter, we give the design and implementation details of our efficient fault tolerant systems. To implement our methods the following messages are used for the communication among the nodes:

- TQ message: a task request sent from one node to others. This message has the form <q, n> where *q* is the requests and *n* is the id of the requesting node.
- QR message: a request together with result sent from a node to the voting center. A QR message can be typically viewed as a message-tuple whose last element has null value.
- QV message: a verdict of a request sent from one node to the others. A QV message is also a message-tuple whose last element is either true or false (no null value).

4.1 The functions of each node

Each node can only execute one request at a time. It maintains a list *W* for all requests which have not yet been executed by this node. The calculated results are saved in a list *C*. List C contains tuples <q, r, t> for all request q whose result r is computed locally at time t but its correctness is unknown. It is used to keep the records of all the executed requests with their associated results so that their correctness may be obtained later on according to Proposition 1 and 2. Note that a request r may appear in either W or C, or both. It may appear in both W and C because the same request may be asked more than once for computation.

In addition to these two lists, a node also maintains a list V for all those requests q whose verdicts are received while their request messages have not yet arrived: Due to network delay or network malfunctions, it is possible for a node to receive a verdict of a request q before it receives the request itself. In this case, q appears neither in W nor C when its verdict arrives. We keep in V such request q, together with its verdict, in order to avoid

the redundant computation of q when the original TQ message arrives.

When a node receives a TQ message, it appends the message to list W. For each request q, after having finished the computation, the node will send the result to the voting center, and add the request/result pair to list C.

When the node gets the correct result for a request *q* from the voter, it checks its C list to see if the result it has calculated for *q* equals to the result returned by the voter.

- If so, it retrieves all the requests it has calculated since its last repair, up until calculating *q*. Based on Proposition 1, it concludes that all these results are correct. In the next step, the node sends a QV message to the voter indicating its results for these requests as correct and then discards them from list C.
- If not, it retrieves all the requests it has calculated after *q* and up until it was repaired. Based on Proposition 2, the node concludes that all these results are incorrect. In the next step, the node sends a QV message to the voter indicating its results for these requests as incorrect and then discards them from list C.

In the case that the node has not yet calculated *q*, it will discard all occurrences of *q* from its W list so it would not calculate *q* any longer: When the correct answer of q is found, there is no need to compute it again. Note that if the node is in the middle of processing *q* when receiving the correct result for it, it will halt the calculation immediately and discard *q*. Then it will go to the next request waiting to be calculated.

The following algorithm shows our method when a verdict *v* for a request *q* is received:

lable1:
if q appears neither in W nor C, add it to V
delete all occurrences of q in W
for each occurrence q' of q in list C
    if R(p,t,q')=v
        for each request q'' computed before q'

send out its result as verdict

delete all occurrences of q'' in W

remove q'' from list C

remove q' from C

else

for each request q'' computed after q'

move q'' from list C to list W

remove q' from C

repair, and then start with label1

4.2 The functions of the voting center

The voting center maintains a list of records, one for each request whose correct result is still unknown.

Since different nodes may produce different results for the same request, each record is associated with a list of results so far received.

The voting center keeps a list Q of triplets <q, r, k> for all requests q whose verdict has not yet been reached. Here q is a request, r is a result, and k is the number of receipts of r as an answer for q.

When a QR message <q,r,t,-> arrives, the voting center checks whether it can make a decision on the correct result of request q, based on the majority vote.

- If so, it will send the verdict (the correct result) to all the nodes. Note that it does not need to record the correct result, since it will not make use of it in the future. Thus, the record of this request is removed afterwards. In addition, all the results for request *q* are deleted from the Q list.
- If it cannot make a decision based on the majority vote, it will only record the result in list Q for future use.

When a QV message <q,r,t,*> arrives, the voting center checks will work as follows:

- If this message <q,r,t,true> indicates that a result for a request *q* is correct, the voter no longer waits for more results for it: Accepting the result in the message, it forwards the result to the user and the other nodes which can process *q*. In the next step, it discards the record regarding *q* and all of its results from its Q list.
- If this message <q,r,t,false> indicates that a result for a request *q* is correct, the voter deletes the result *r* from its Q list.

The following example illustrates how our implemented fault tolerant method works:

Example 2: In Example 1, we have illustrated the major steps in our method. Now we show its implementation details: how the data structures used in our method are used and updated in each step of Example 1.

In the second step, when nodes N1, N2, N3, N4, N5 and N6 receive requests sent in step 1, their W list looks as follows:

Node	**W list**
N1	[<q_5,N6>,<q_3,N4>,<q_8,N6>,<q_5,N2>,<q_6,N5>,<q_1,N2>,<q_6,N3>,<q_1,N3>,<q_7,N2>,<q_7,N4>,<q_8,N3>]
N2	[<q_1,N1>,<q_8,N6>,<q_3,N4>,<q_3,N1>,<q_7,N4>,<q_6,N5>,<q_5,N6>,<q_4,N1>,<q_1,N3>,<q_8,N3>,<q_6,N3>]
N3	[<q_1,N1>,<q_1,N2>,<q_3,N4>,<q_7,N4>,<q_3,N1>,<q_5,N2>,<q_4,N1>,<q_5,N6>,<q_8,N3>,<q_6,N3>,<q_7,N2>]
N4	[<q_8,N3>,<q_1,N1>,<q_4,N1>,<q_3,N1>,<q_7,N2>,<q_6,N3>,<q_5,N6>,<q_1,N2>,<q_5,N2>,<q_8,N6>,<q_6,N5>, <q_1,N3>]
N5	[<q_5,N6>,<q_3,N4>,<q_1,N1>,<q_4,N1>,<q_7,N2>,<q_6,N3>,<q_8,N6>,<q_1,N3>,<q_3,N1>,<q_7,N4>,<q_6,N3>, <q_1,N2>]
N6	[<q_1,N1>,<q_3,N4>,<q_8,N3>,<q_1,N2>,<q_5,N2>,<q_7,N2>,<q_4N1>,<q_1,N3>,<q_6,N3>,<q_3,N1>,<q_7,N4>]

Table 3

In step 3, when the nodes have finished the calculation of a number of requests and have sent the results to the voter, their W and C list will look as follows:

Node	**W list**	**C list**

N1	[<q_5,N2>,<q_6,N5>,<q_1,N2>,<q_6,N3>,<q_1,N3>,<q_7,N2>,<q_7,N4>,<q_8,N3>]	[<q_5,r_5,t_1>,<q_3,r_3,t_2>,<q_8,r_8,t_3>]
N2	[<q_3,N1>,<q_7,N4>,<q_6,N5>,<q_5,N6>,<q_4,N1>,<q_1,N3>,<q_8,N3>,<q_6,N3>]	[<q_1,r_1,t_1>,<q_8,r_8,t_3><q_3,r_3,t_5>]
N3	[<q_7,N4>,<q_3,N1>,<q_5,N2>,<q_4,N1>,<q_5,N6>,<q_8,N3>,<q_6,N3>,<q_7,N2>]	[<q_1,r_1,t_2>,<q_3,r_3,t_3>,<q_3,r_3,t_4>]
N4	[<q_3,N1>,<q_7,N2>,<q_6,N3>,<q_5,N6>,<q_1,N2>,<q_5,N2>,<q_8,N6>, <q_6,N5>, <q_1,N3>]	[<q_8,r_{80},t_2>,<q_1,r_{10},t_1>,<q_4,r_{40},t_3>]
N5	[<q_7,N2>,<q_6,N3>,<q_8,N6>,<q_1,N3>,<q_3,N1>,<q_7,N4>,<q_6,N3>, <q_1,N2>]	[<q_5,r_5.t_1>,<q_1,r_1,t_3>,<q_3,r_3,t_6>, <q_4,r_4,t_8>]
N6	[<q_1,N2>,<q_5,N2>,<q_7,N2>,<q_4N1>,<q_1,N3>,<q_6,N3>,<q_3,N1>,<q_7,N4>]	[<q_1,r_{10},t_1>,<q_3,r_{30},t_2>,<q_4,r_{40},t_3>]

Table 4

In step 4, if we consider that the voter receives the results in step 3 in the following order:

<q_5,r_5,t_1,->,<q_1,r_1,t_1,->,<q_8,r_{80},t_1,->,<q_1,r_1,t_2,->,<q_1,r_{10},t_1,->,<q_5,r_5,t_1,->,

<q_3,r_3,t_2,->,<q_8,r_8,t_3,->,<q_1,r_1,t_3,->,<q_3,r_{30},t_2,->,<q_1,r_{10},t_2,->,<q_3,r_3,t_3,->,

<q_3,r_3,t_4,->,<q_8,r_8,t_3,->,<q_3,r_3,t_5,->,<q_4,r_{40},t_3,->,<q_4,r_{40},t_3,->,<q_4,r_4,t_8,->,<q_1,r_1,t_6,->,

the voter's Q list will evolve as follows in step 5. In each step of the following table, one of the requests is processed in the order it is received (as presented above). The correct result found for a request in each step is shown as well.

Step	**Q list**	**Correct result found**
1	<q_5,r_5,t_1>	No result
2	<q_5,r_5,t_1>,<q_1,r_1,t_1>	No result
3	<q_5,r_5,t_1>,<q_1,r_1,t_1>,<q_8,r_{80},t_1>	No result
4	<q_5,r_5,t_1>,<q_1,r_1,t_1>,<q_8,r_{80},t_1>,<q_1,r_1,t_2>	No result
5	<q_5,r_5,t_1>,<q_1,r_1,t_1>,<q_8,r_{80},t_1>,<q_1,r_1,t_2>,<q_1,r_{10},t_1>	No result
6	<q_5,r_5,t_1>,<q_1,r_1,t_1>,<q_8,r_{80},t_1>,<q_1,r_1,t_2>,<q_1,r_{10},t_1>,<q_5,r_5,t_1>	No result
7	<q_5,r_5,t_1>,<q_1,r_1,t_1>,<q_8,r_{80},t_1>,<q_1,r_1,t_2>,<q_1,r_{10},t_1>,<q_5,r_5,t_1>,<q_3,r_3,t_2>	No result
8	<q_5,r_5,t_1>,<q_1,r_1,t_1>,<q_8,r_{80},t_1>,<q_1,r_1,t_2>,<q_1,r_{10},t_1>,<q_5,r_5,t_1>,<q_3,r_3,t_2>,<q_8,r_8,t_3>, <q_1,r_1,t_3>,<q_3,r_{30},t_2>	No result
9	<q_5,r_5,t_1>,<q_1,r_1,t_1>,<q_8,r_{80},t_1>,<q_1,r_1,t_2>,<q_1,r_{10},t_1>,<q_5,r_5,t_1>,<q_3,r_3,t_2>,<q_8,r_8,t_3>, <q_1,r_1,t_3 >,<q_3,r_{30},t_2>,<q_1,r_{10},t_2>	No result
10	<q_5,r_5,t_1>,<q_1,r_1,t_1>,<q_8,r_{80},t_1>,<q_1,r_1,t_2>,<q_1,r_{10},t_1>,<q_5,r_5,t_1>,<q_3,r_3,t_2>,<q_8,r_8,t_3>, <q_1,r_1,t_3>,<q_3,r_{30},t_2>,<q_1,r_{10},t_2>,<q_3,r_3,t_3>	No result

11	$<q_5,r_5,t_1>,<q_1,r_1,t_1>,<q_8,r_{80},t_1>,<q_1,r_1,t_2>,<q_1,r_{10},t_1>,<q_5,r_5,t_1>,<q_3,r_3,t_2>,<q_8,r_8,t_3>,$ $<q_1,r_1,t_3>,<q_3,r_{30},t_2>,<q_1,r_{10},t_2>,<q_3,r_3,t_3>,<q_3,r_3,t_4>$	No result
12	$<q_5,r_5,t_1>,<q_1,r_1,t_1>,<q_8,r_{80},t_1>,<q_1,r_1,t_2>,<q_1,r_{10},t_1>,<q_5,r_5,t_1>,<q_3,r_3,t_2>,<q_8,r_8,t_3>,$ $<q_1,r_1,t_3>,<q_3,r_{30},t_2>,<q_1,r_{10},t_2>,<q_3,r_3,t_3>,<q_3,r_3,t_4>,<q_8,r_8,t_3>$	No result
13	$<q_5,r_5,t_1>,<q_1,r_1,t_1>,<q_8,r_{80},t_1>,<q_1,r_1,t_2>,<q_1,r_{10},t_1>,<q_5,r_5,t_1>,<q_3,r_3,t_2>,<q_8,r_8,t_3>,$ $<q_1,r_1,t_3>,<q_3,r_{30},t_2><q_1,r_{10},t_2>,<q_3,r_3,t_3>,<q_3,r_3,t_4>,<q_8,r_8,t_3>,<q_3,r_3,t_5>$	r_3 correct result of q_3
14	$<q_5,r_5,t_1>,<q_1,r_1,t_1>,<q_8,r_{80},t_1>,<q_1,r_1,t_2>,<q_1,r_{10},t_1>,<q_5,r_5,t_1>,<q_8,r_8,t_3>,<q_1,r_1,t_3>,$ $<q_1,r_{10},t_2>,<q_8,r_8,t_3>,<q_4,r_{40},t_3,->$	No result
15	$<q_5,r_5,t_1>,<q_1,r_1,t_1>,<q_8,r_{80},t_1>,<q_1,r_1,t_2>,<q_1,r_{10},t_1>,<q_5,r_5,t_1>,<q_8,r_8,t_3>,<q_1,r_1,t_3>,$ $<q_1,r_{10},t_2>,<q_8,r_8,t_3>,<q_4,r_{40},t_3,->,<q_4,r_{40},t_3>$	No result
16	$<q_5,r_5,t_1>,<q_1,r_1,t_1>,<q_8,r_{80},t_1>,<q_1,r_1,t_2>,<q_1,r_{10},t_1>,<q_5,r_5,t_1>,<q_8,r_8,t_3>,<q_1,r_1,t_3>,$ $<q_1,r_{10},t_2>,<q_8,r_8,t_3>,<q_4,r_{40},t_3>,<q_4,r_{40},t_3>,<q_4,r_4,t_8>$	No result
17	$<q_5,r_5,t_1>,<q_1,r_1,t_1>,<q_8,r_{80},t_1>,<q_1,r_1,t_2>,<q_1,r_{10},t_1>,<q_5,r_5,t_1>,<q_8,r_8,t_3>,<q_1,r_1,t_3>,$ $<q_1,r_{10},t_2>,<q_8,r_8,t_3>,<q_4,r_{40},t_3>,<q_4,r_{40},t_3>,<q_4,r_4,t_8>,<q_1,r_1,t_6>$	r_1 correct result of q_1
18	$<q_5,r_5,t_1>,<q_8,r_{80},t_1>,<q_5,r_5,t_1>,<q_8,r_8,t_3>,<q_8,r_8,t_3>,<q_4,r_{40},t_3>,<q_4,r_{40},t_3>,<q_4,r_4,t_8>$	

Table 5

In step 6, when the nodes receive the verdicts regarding requests q_1 and q_3 from step 5, their W and C list will change as follows. The column "Deleted from W list" shows the requests which should not be calculated any more and the column "Deleted from C list" shows the results whose correctness or incorrectness have been found by using Proposition 1 and 2.

Nodes	**Changed W list**	**Deleted from W list**	**Changed C list**	**Deleted from C list**
N1	$[<q_6,N5>,<q_6,N3>,$ $<q_7,N2>,<q_7,N4>,<q_8,N3>]$	$<q_1,N2>,<q_1,N3>,$ $<q_5,N2>$	$<q_8,r_8,t_3>$	$<q_5,r_5,t_1>,<q_3,r_3,t_2>$
N2	$[<q_7,N4>,<q_6,N5>,<q_5,N6>,$ $<q_4,N1>,<q_6,N3>]$	$<q_3,N1>,<q_1,N3>,$ $<q_8,N3>$	empty	$<q_1,r_1,t_1>,<q_8,r_8,t_3><$ $q_3,r_3,t_5>$
N3	$[<q_7,N4>,<q_5,N2>,<q_4,N1>,<q_5,N$ $6>,<q_8,N3>,<q_6,N3>,<q_7,N2>]$	$<q_3,N1>$	empty	$<q_1,r_1,t_2>,<q_3,r_3,t_3>,<$ $q_3,r_3,t_4>$
N4	$[<q_7,N2>,<q_6,N3>,<q_5,N6>,$ $<q_5,N2>,<q_8,N6>,<q_6,N5>]$	$<q_3,N1>,<q_1,N2>,$ $<q_1,N3>$	$<q_8,r_{80},t_2>$	$<q_1,r_{10},t_1>,<q_4,r_{40},t_3>$
N5	$[<q_7,N2>,<q_6,N3>,<q_8,N6>,$ $<q_7,N4>,<q_6,N3>]$	$<q_1,N3>,<q_3,N1>,$ $<q_1,N2>$	$<q_4,r_4,t_8>$	$<q_5,r_5.t_1>,<q_1,r_1,t_3>,$ $<q_3,r_3,t_6>,$
N6	$[<q_5,N2>,<q_7,N2>,<q_4N1>,<q_6,N$ $3>,<q_7,N4>]$	$<q_1,N2>,<q_1,N3>,$ $<q_3,N1>$	empty	$<q_1,r_{10},t_1>,<q_3,r_{30},t_2>,$ $<q_4,r_{40},t_3>$

Table 6

In step 7, when the voter receives the verdicts for q_5 and q_8 from step 6, the Q list of the voter will be updated as follows. The column "Deleted from the Q list" shows the results which have been deleted from the Q list because their correctness has been found by the received verdicts:

Changed Q list	**Deleted from the Q list**
$<q_4,r_{40},t_3>,<q_4,r_{40},t_3>,<q_4,r_4,t_8>$	$<q_5,r_5,t_1>,<q_8,r_{80},t_1>,<q_5,r_5,t_1>,<q_8,r_8,t_3>,<q_8,r_8,t_3>$

Table 7

When the voter forwards the verdict for q_5 and q_8 to the other nodes in the last step, the W list and C list of the nodes are updated as follows:

Nodes	**Changed W list**	**Deleted from W list**	**Changed C list**	**Deleted from C list**
N1	$[<q_6,N5>,<q_6,N3>,$ $<q_7,N2>,<q_7,N4>]$	$<q_8,N3>$	empty	$<q_8,r_8,t_3>$
N2	$[<q_7,N4>,<q_6,N5>,$ $<q_4,N1>,<q_6,N3>]$	$<q_5,N6>$	empty	nothing
N3	$[<q_7,N4>,<q_4,N1>,<q_5,N6>,$ $<q_6,N3>,<q_7,N2>]$	$<q_5,N2>,<q_8,N3>$	empty	nothing
N4	$[<q_7,N2>,<q_6,N3>,<q_6,N5>]$	$<q_5,N6>,<q_8,N6>,$ $<q_5,N2>$	empty	$<q_8,r_{80},t_2>$
N5	$[<q_7,N2>,<q_6,N3>,<q_7,N4>,$ $<q_6,N3>]$	$<q_8,N6>$	$<q_4,r_4,t_8>$	nothing
N6	$[<q_7,N2>,<q_4N1>,<q_6,N3>,<q_7,N4>]$	$<q_5,N2>$	empty	nothing

Table 8

Chapter 5. Method Evaluation

Suppose we have n requests $q_1,...,q_n$, each sent to m nodes.

Let

$$A(r_1...r_n) = \{((r_{1,1}...r_{1,n}),...,((r_{n,1}...r_{n,n}))\mid r_{i,1}...r_{i,n} \text{ is a permutation of requests } r_1...r_n \text{ at node } i\}$$

Each element of A is called an *order assignment*. It refers to the order of the requests at each node.

Given an *order assignment* α, a *result assignment* γ_α associates a Boolean value to each request q in α, representing whether the result of q computed at this node is correct or not.

Let $S_{\alpha,\gamma\alpha}$ denote the time saving of the computation of the requests with *order assignment* α and *result assignment* γ_α compared to computing the n requests in synchronized order.

The average saving S(m,n) of using our method compared to computing the n requests in synchronized order is calculated as the mean of the savings with respect to all *order assignment* α and *result assignment* γ_α:

$$S(m,n) = \frac{\sum_{\alpha,\gamma\alpha} S_{\alpha,\gamma\alpha}}{NumOfPermutationsWithAssign}$$

Here *NumOfPermutationsWithAssign* denote the total number of permutations of the requests that $q_1,...,q_n$ can have in m nodes, combined with all possible *result assignments* for each permutation.

Clearly, a precise measurement of the savings will take too much time to calculate due to the exponential blowout caused by the total number of permutations of the requests and the total number of *result assignments* for each permutation.

Thus, we have randomly selected some samples to evaluate the performance of our method.

5.1 Evaluation settings

We have evaluated our method by means of simulation. The main focus of our experiments is to determine the saving of our method in the following two aspects:

1. saving of the time we use to reach a correct result;
2. reduction of the number of redundant computations we need to perform.

There are many factors that affect the performance of our method, for example:

a) number of nodes,
b) maximum number of faulty nodes within certain time period considered,
c) number of times that a same request is repeatedly requested,
d) the number of requests a node can execute in parallel,
e) probability of a node becoming faulty,
f) repair time,
g) total number of requests within the time period considered,
h) the time each request takes for computation,
i) time for message passing,
j) etc.

Among all these factors having effects on the performance of our method, we will consider the first three as parameters.

a) The *number of nodes* is of great importance: both the *time consumed to reach the correct result* of a request and the *number of redundant executions of a request* have direct dependencies to it.

- The estimation of the time saved to reach the correct result of a request q is dependent on the number of nodes which have not yet calculated q. Therefore

when the total number of nodes changes, the number of nodes which have not yet calculated q is changed as well. Because of this, the performance of our method is affected.

- The number of redundant computations of a request q which can be avoided depends on the number of nodes which have not yet calculated q as well. Therefore the change in the total number of nodes changes the number of nodes which have not yet calculated q, consequently affecting the performance of our method.

These two phenomena are thoroughly investigated in the following sections.

b) The *maximum number of faulty nodes* is highly important as well, since this number directly effects both the *computation time* and the *number of answers needed* to find the correct result for a request.

- When the maximum number of faulty nodes changes, the *policy* used by the voter to find the correct result of a request changes as well. By *policy* we mean the number of common results needed by the voter so it can choose the correct result of a request. This change affects the performance of our method.
- When the number of faulty nodes changes, the number of nodes which can use Proposition 1 is changed as well, consequently affecting the performance of our method.

c) The *number of times that a same request is repeatedly requested* can affect the *number of redundant computations we need to perform*. The amount of reduction in the redundant computations of a request q is dependent on the number of occurrences of request q not calculated in the system. When a request is sent repeatedly, the number of occurrences of request q changes, consequently affecting the *number of redundant computations we need to perform*.

d) Different nodes can have different degrees of parallelism when it comes to executing requests. This also has impact on the *time we use to reach a correct result*. To simplify

the evaluation of our method, we only consider sequential execution: each node can execute only one request at a time.

e) Different nodes may have different probability to turn faulty. For simplicity, we consider that all nodes have the same probability. Based on real life experience, we sometimes adopt a system administration rule which demands for the removal of any machine that shows faulty behavior more than 50% of the time. Therefore we consider that all nodes may become faulty with a probability of 50%, which is high enough.

f) We consider that each node takes the same amount of time to get repaired.

g) We consider that all the nodes send the same number of requests. Each node can send a request at any time it desires.

h) The time spent on computing each individual result directly effects the time we use to reach a (global) correct result. For simplicity, we consider that the amount of computation time needed for processing each request in each node is the same.

g) The time spent on message passing also directly effects the time we use to reach a correct result. For simplicity, we assume that the time consumed from the moment a request is sent into the network to the time it reaches its destination is the same for all requests. This amount of time is considered as *one time unit* in the simulations.

Now we show the performance evaluation of our method. Section 5.1 is about the evaluation of the amount of computation saving (Section 5.1.1) and the amount of time saving (Section 5.1.2). Section 5.2 illustrates the trade-off.

5.1.1 Evaluating the amount of computation saving

In this section we show how much our method can reduce the *number of redundant computations of a request*. We have presented *three evaluations* to show the performance

of our method:

a) when the total number of nodes *TN* changes;
b) when the number of faulty nodes *FN* changes;
c) when the *number of times RT that a same request is repeatedly requested* changes.

Our method can reduce the number of redundant computations in the situation when a node receives the correct result for a request q before or during its own calculation of q. In this case, the node will discard q without calculating it any longer. The reason behind is that when the correct result for a request is found, there is no need for it to be calculated by other nodes any longer. Therefore, all remaining calculations of q can be eliminated. These eliminations reduce the number of redundant computations of a request.

Evaluation when TN changes

The following diagram shows the number of redundant computations avoided for a request while the number of nodes changes. In this evaluation we have kept the values of the parameters *FN* and *RT* fixed while the value of *TN* changes. In the following diagram FN = 49%, RT = 0 while each node sends 15 distinct requests throughout the simulation run. TN ranges from 10 nodes to 400 nodes. Since each node can become faulty at anytime, and the nodes can send out requests with any pace, each data in the following diagram is collected as the average of 100 simulation runs.

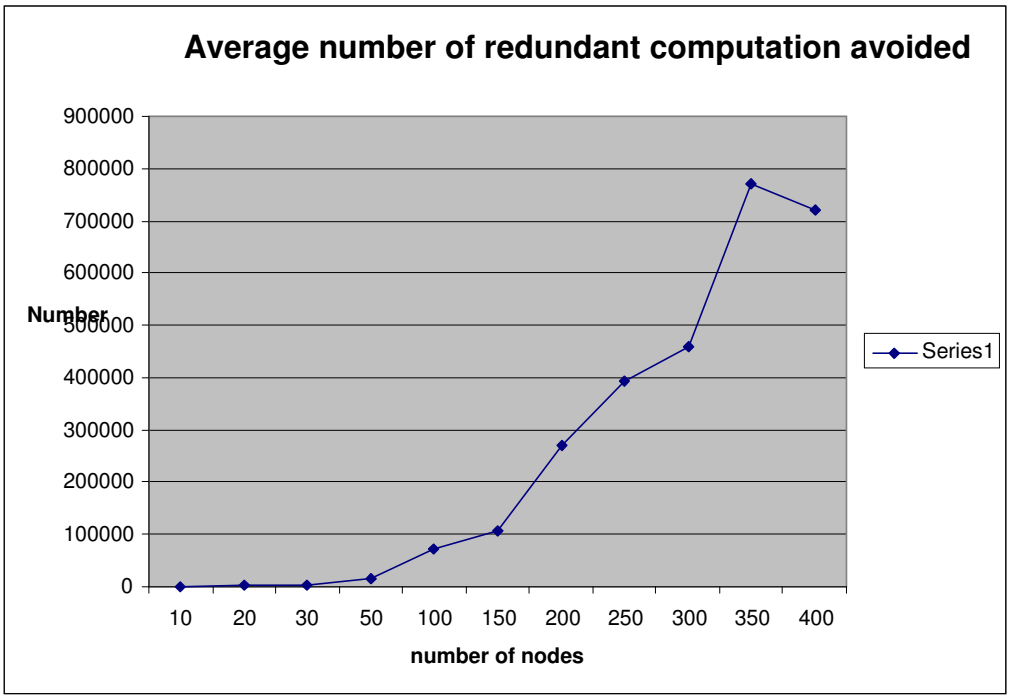

Figure 7 Number of redundant computations avoided in the system

It can be seen that *the number of redundant computations avoided increases with the increase of the nodes*. The number of computation saving starts from 585 and goes up to 2700, 4320,…, until 720000. This result is not so strange, because a system with a larger number of nodes will have more nodes with computation saving compared to a system with a smaller number of nodes, in a same situation. In other words, when there are more nodes active in the system, a computation can be avoided in more nodes, in the sense that when the voter finds the correct result to a request, there are more nodes in which the request can be simply ignored. For example, when the voter chooses the correct result for request *q*, 10% of the total nodes have sent in the wrong result and 50% have sent in the correct one. In this case redundant computation is avoided in the remaining 30% of the nodes which have not yet sent in a result. Consequently, in a system with 100 nodes, the number of computations avoided is 30, while in a system with 200 nodes this value is 60.

In addition to the above observation, we can also see that the *increase of the computation saving is linear to the increase of the TN*. This is shown in the following diagram.

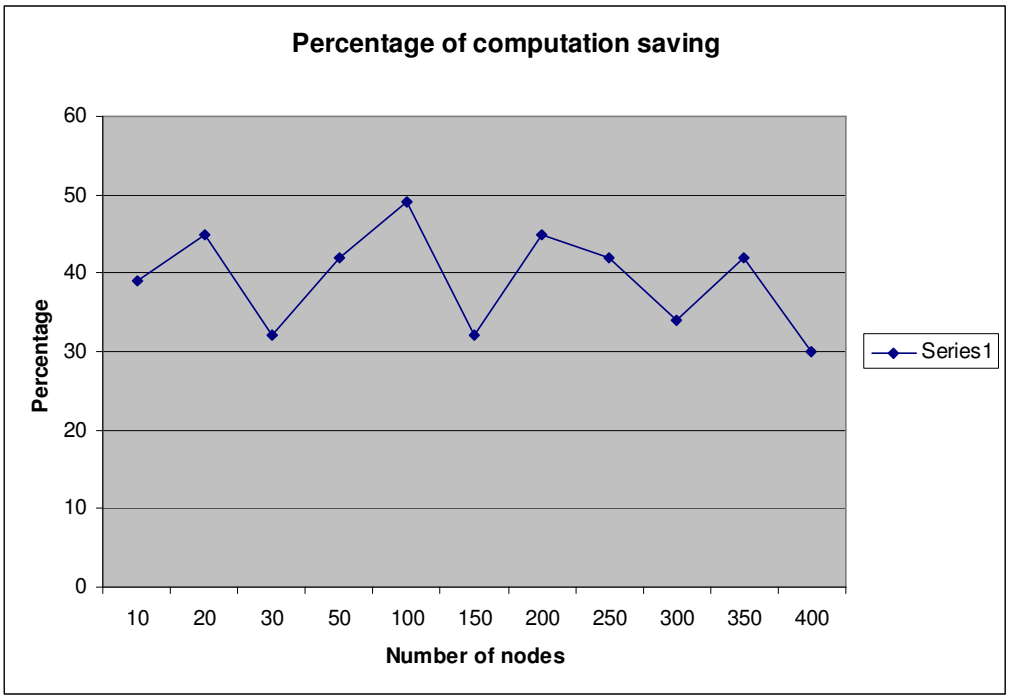

Figure 8 Percentage of computation saving

Here, the percentage (computation) saving is defined by the following formula:

$$\frac{Total\ number\ of\ \ requests\ sent\ in\ the\ system}{Total\ number\ of\ \ redundant\ computations\ avoided} = \frac{100}{x}$$

Our evaluations show an approximately 39% computation saving for any TN. We give our intuitive explanation about it below.

A voter can have any of the situation S_i =(a,b,c) for each request, where *a* is the percentage of wrong results; *b* is the percentage of correct results, and *c* is the percentage of computation savings. According to our assumption, b = 50% (for majority vote), and thus we also have a < 50%, c < 50%. Considering only integer numbers, we have 50 different situations, e.g. (1%, 50%, 49%), (2%, 50%, 48%), ... (49%, 50%, 1%). Let p_i be the probability of situation S_i with constraint $\sum_{i=1}^{number\ of\ situations} p_i = 1$. For simplicity, we consider the same probability for all the situations, i.e. $\forall$i, p_i = 1/50.

Let c(S) denote the third element of situation S. The *predicted average percentage saving*

is

predicted average percentage saving (paps)

$= [p_1 * c(S_1) + p_2 * c(S_2) + \ldots + p_n * c(S_n)]$

$= p_1 * [c(S_1) + c(S_2) + \ldots + c(S_n)]$

$= 1/50 * [50\% + 49\% + \ldots + 40\% + \ldots + 30\% + \ldots + 20\% + \ldots + 10\% + \ldots + 1\%]$

$= 24\%.$

Now that we know the average percentage of computation saving *of one request*, the average computation saving of *the total system x* can be achieved by:

$$\frac{\sum_{n=1}^{\text{total number of nodes}} \text{number of requests sent by node } n}{\text{paps} \times \sum_{n=1}^{\text{total number of nodes}} \text{number of requests sent by node } n} = \frac{100}{x}$$

As we have *paps*=24%, it is clear that x = 24.

We have observed an average percentage saving (39%) higher than predicted (24%). This is because of the use of Proposition 1 which helps the voter to find the correct results in a better way.

Evaluation when FN changes

The following diagram shows the amount of redundant computations avoided for a request while the number of faulty nodes changes. In this evaluation we have kept the values of the parameters TN and RT fixed while the value of FN changes. In the following diagram TN = 200, RT = 0 while each node sends 15 distinct request throughout the simulation run. FN ranges from 5% to 49% of the total nodes. Each node becomes faulty with a probability of 50%. Since each node can become faulty at anytime, and nodes can send out requests with any random pace, each data in the following diagram is collected as the average of 100 simulation runs.

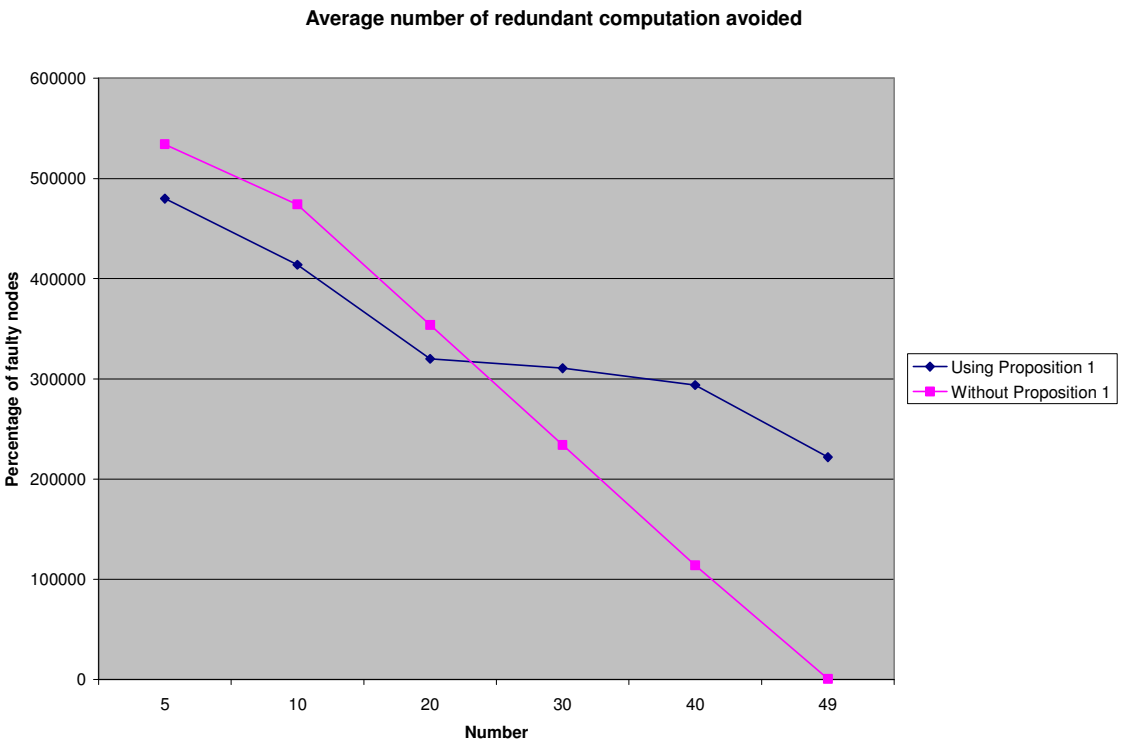

Figure 9 Average number of redundant computation avoided

As it can be seen, *with the increase of the percentage of faulty nodes, the number of avoided redundant computations decreases,* e.g., the system with 5% faulty nodes has the most computation saving of 480000 computations. We give our intuitive explanation below.

Faulty results in voter:	5%...10%…20%…30%…40%…49%
Correct results in voter:	6%…11%…21%…31%…41%…50%
Remaining nodes:	89%…79%…59%…39%…19%…1%

Figure 10 Voter functionality

When a maximum of 49% of the nodes can become faulty at any time, at least 50% of the non-faulty nodes are needed to produce a result for a request q so that the voter can find its correct result. Compared to the situations with less percentage of faulty nodes, since *more nodes* are needed to calculate a result for q, the number of computations which can be avoided decreases. It can be seen in picture 10 that with the increase of the percentage

of faulty nodes, the percentage of nodes which have not yet calculated *q* drops from 89% to 1%.

In addition, since all the nodes have the same computation power and process requests sequentially, the *time* needed for 50% of non-faulty nodes to produce results is most probably larger compared to the time needed for 6%, 11%, 21%, 31% or 41% of non-faulty nodes to produce results. Consequently, since a longer time is consumed until the voter can find the correct result for *q*, a smaller number of nodes (1%) can be exempted from calculating it.

Another factor that affects the computation saving is the application of Proposition 1. We explain below that *with the increase of faulty nodes, Proposition 1 will cause less computation saving*. With the increase of faulty nodes, since the result of *q* is achieved in a longer time, Proposition 1 triggered by the verdict regarding *q* is executed later as well. For the same reason mentioned above, for each request *q'* whose correct result can be found by this Proposition, it is more probable that a larger number of nodes have the chance to calculate it. Therefore, computation of *q'* is avoided in a smaller number of nodes, resulting in an increase in computation saving.

Note that this phenomenon is common to all cases when we use any voting algorithms (not necessarily Proposition 1), as shown in Figure 10 for both the experiments with and without using Proposition 1.

However, compared to the similar experiment but using solely simple voting (without Proposition 1), another observation from our experiment is that *the decrease of computation saving slows down with the increase of the allowed faulty nodes.* This phenomenon is due to the use of Proposition 1 in our method. Proposition 1 allows the non-faulty nodes to find the correct result of a request before the voter can do so. Therefore, it can aid in the increase of the number of exempted computations, slowing the decrease of this number.

Evaluation when RT changes

Recall that a request can be repeatedly sent, the present evaluation shows how the performance of our method is affected by the change of the repeated requests: The following diagram shows the amount of redundant computations avoided while the number of repeated requests changes. In this evaluation we have kept the values of the parameters TN and FN fixed while the value of RT changes. In the following diagram, TN = 200, FN = 49% while each node sends 15 requests throughout the simulation run. RT ranges from 5 to 15. The value of RT shows the number of common requests each node shares with the other nodes. Since each node can become faulty at anytime, and nodes can send out requests with any random pace, each data in the following diagram is the average of 100 simulation runs.

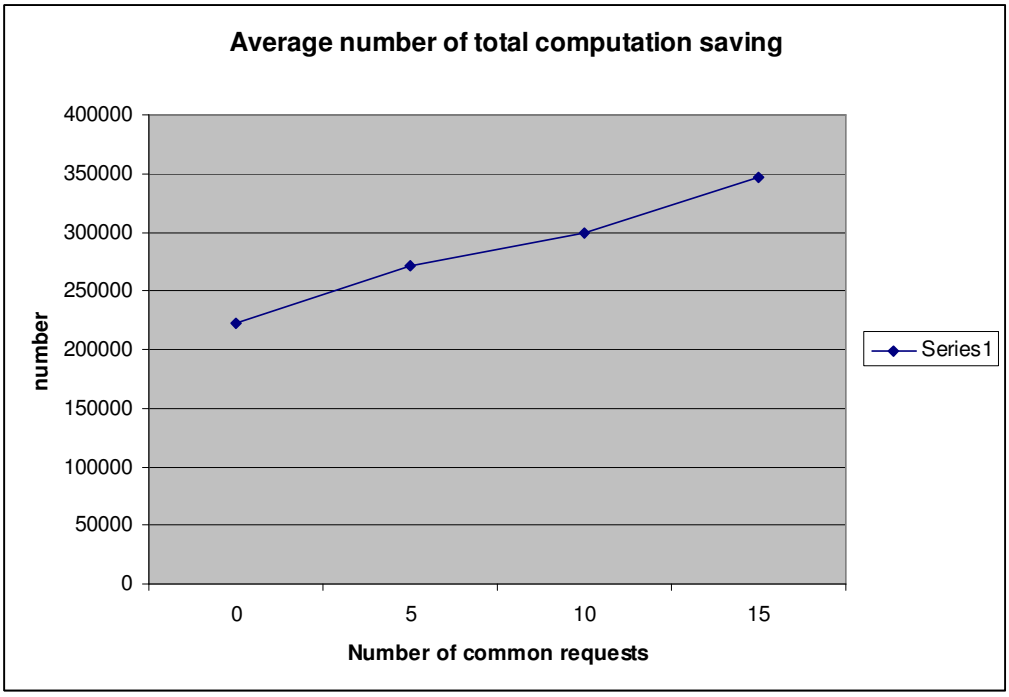

Figure 11 Average number of total computation saving

It can be seen that, with the increase of RT, the number of computation saving increases as well. The reason for this increase is that when a request q is sent out repeatedly, the number of q requests existent in the W list of nodes increases as well. Therefore when the result of q is found by the voter or by Proposition 1, there will be more q requests in the

W list, compared to the situation where *q* is sent out only once. Thus, when more requests are sent out repeatedly, more computation saving is achieved.

5.1.2 Evaluating the amount of time saving

Note that our method can reduce the time to reach a correct result due to the following reasons:

- Using Proposition 1, we may be able to conclude that the results it has calculated for a number of requests are correct, before getting their correct results from the voting center. This situation speeds up the time to find the correct result for a request.
- Due to the use of Proposition 1, the voter can avoid waiting for a sufficient number of results for a request *q* so it can be able to choose the correct result for it. In this case an unnecessary waiting time is eliminated, resulting in the reduction of the time consumed to reach the correct result.

In this section we show how much our method can reduce the time consumed for the correct result of a request to be achieved. We present three evaluations to show the performance of our method.

Note that *time saving* is a different concept from *computation saving*. *Time saving* refers to the reduction of the time consumed for reaching a correct result of a request, while *computation saving* refers to the number of redundant computations avoided while still guaranteeing fault tolerance. In other words, if the average time consumed to reach a correct result in a system without making use of Proposition 1 and 2 is *r,* and the average time consumed to reach a correct result with our method is *r'*, then the amount of time saving is *r-r'*. The estimation of time saving can be achieved by the following steps:
When a Proposition 1 finds the correct result for a request *q*

1. Derive the amount of time consumed by *each* non-faulty node for it to be able to calculate a result for *q*. This time can be found by the following function:

$$Time\ Consumed\ in\ a\ node\ (TCN) := \begin{cases} index\ of\ the\ cell\ containing\ q\ in\ W\ list \times computation\ time & if\ q\ exist\ in\ W \\ remaining\ computation\ time\ of\ q & if\ q\ is\ currently\ being\ executed \\ length\ of\ list\ W \times computation\ time & if\ q\ has\ not\ yet\ been\ received \\ 0 & if\ q\ has\ been\ executed \end{cases}$$

The amount of time consumed by a node to calculate request q can be estimated in one of the following ways:

I. *if the node has not yet calculated q and it is in its W list*: check how many requests in the W list it has to calculate until it reaches q. This number of requests can be found by the index of the cell that contains request q (ind_q) in list W. The time t consumed to finish the calculation of these requests can be found by multiplying ind_q with the time consumed to calculate each request (*computation time*). Recall that all requests have the same *computation time.*

II. *if the node is calculating q when receiving a verdict regarding q*: The remaining computation time needed to complete the execution of q is the time saved.

III. *if the node has not yet received q*: We assume that the next request that the node will receive is q. Therefore the amount of time consumed so the node can be able to calculate q, is the current number of requests in the W list multiplied by the *computation time*. This number of requests is equal to the current length of the W list. Note that the W list works like a vector, and therefore its length varies with the number of requests residing in it.

IV. *if the node has calculated q*: No time is consumed because it has already been calculated.

2. Among those with $TCN \neq 0$, we sort the *TCN* of all the non-faulty nodes in an

increasing order and put them in a list called *TC*.

3. check how many correct results the voter still needs to be able to find the correct result for *q*. This number can be derived with the following formula:

   if we consider:

   - *cR* as the number of requests which have sent back the correct result to the voter,
   - *faultLimit* as the maximum number of faulty nodes, and
   - *rN* as the number of correct results needed for the voter to find the correct result,

   then

   $$rN = faultLimit - cR+1$$

   The voter needs *faultLimit+1* number of common results to be able to declare it as the correct result. At this stage the voter compares the results it has received with the correct result *r* found by Propostion 1. It sees that *cR* number of nodes have sent back the value *r*. In this situation, if the system did not use Proposition 1, the voter needed *faultLimit - cR+1* more results with the value of *r* to be able to declare it as correct.

4. in the last step, the time saved to reach the correct result of *q* is calculated as follows: As we mentioned, the voter still needs *rN* number of requests to be able to declare *r* as the correct result. To find the average amount of time *wt* the voter has to wait to receive the remaining *rN* results, we first extract *rN* number of *TCNs* from the beginning of the *TC* list. To find the value of *wt* we will find the possible maximum and minimum time the voter needs to wait, and consider their median value as *wt*.

   - maximum amount of wait (*Maxwt*): if we consider that all the *rN* number of nodes chosen from the *TC* list calculate their request in W list sequentially , then:

     Maxwt = 0

     for the first rN number of TCNs in the list TC

$$Maxwt = Maxwt + TCN$$

- Minimum amount of wait (*Minwt*): if we consider that all the *rN* number of nodes calculate their requests at the same time then *Minwt* will equal the value of the *rN*'th *TCN* in the *TC* list.

Therefore $wt = (Minwt+Maxwt)\div 2$. *wt* represents the average amount of time saved to reach the correct result of *q* because the voter does not need to wait for an additional *wt* amount of time to find the correct result any longer. Please note that in calculating the time saving, the time consumed for the results to be sent to the voter is exempted.

Evaluation when TN changes

The following diagram shows the average time saved in reaching the correct results of the requests while the total number of nodes changes. In this evaluation we have kept the values of the parameters FN and RT fixed while the value of TN changes. In the following diagram FN = 49%, RT = 0 while each node sends 15 distinct request throughout the simulation run. TN ranges from 10 nodes to 400 nodes. Since each node can become faulty at anytime, and the nodes can send out requests with any pace, each data in the following diagram is the average of 100 simulation runs.

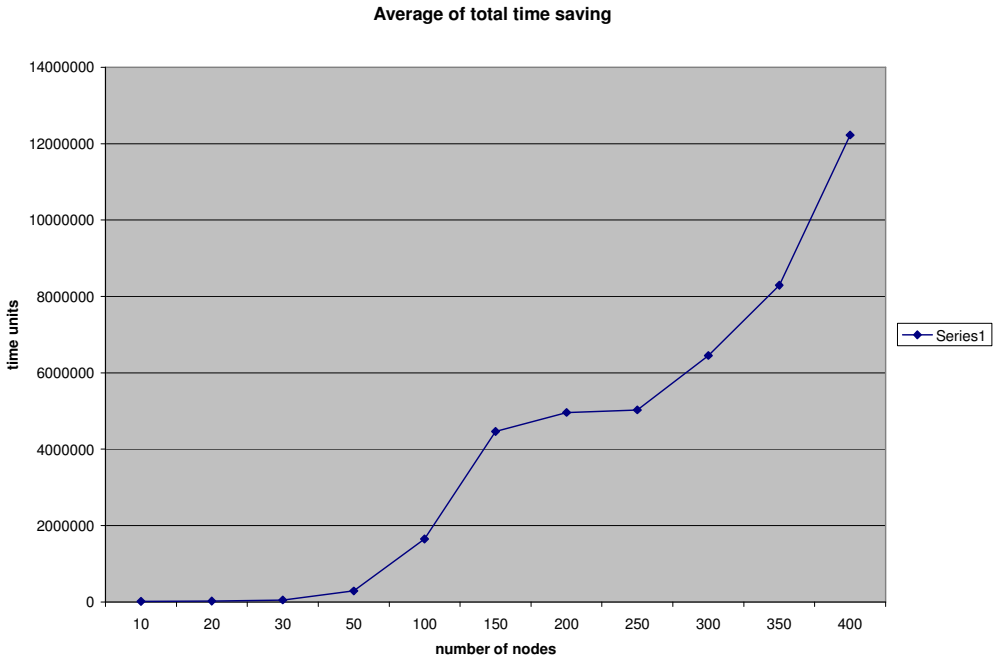

Figure 12 Average of total time saving

Each data in the presented diagram represents the value of *wt*. As it can be seen, with the increase of the number of nodes, the time saved to reach a correct result increases. The increase can be explained by the fact that with the increase of the number of nodes, when Proposition 1 finds the correct result for a request, there will be more nodes contributing to time saving. For example, consider two systems, one with 100 nodes and the other with 200 nodes, while in both systems 10% of the nodes are faulty. In both systems a situation is reached where Proposition 1 finds the correct result *r* for request *q*, while the voter has only received 10% of the total number of results it needs to declare *r* as the correct value. In this situation, the time saving in the system with 100 nodes is:

Number of common results the voter needs to make a decision: 10% of 100 nodes = 10,

10% of 10 result has been received by the voter = 1

Number of results still needed by the voter: 10-1+1 = 10

The time saving is: Maxwt = $\sum_{i=1}^{10} TC[i]$ and Minwt = TC[10]. In the worst case, all the *TCN* values in the *TC* list equal 1, then Maxwt = 10 and Minwt = 1. Therefore

wt = 5.5.

On the other hand the time saving for the system with 200 nodes is:

Number of common results the voter needs to make a decision: 10% of 200 nodes = 20,

10% of 20 result has been received by the voter = 2

Number of results still needed by the voter: 20-2+1 = 19

The time saving is: Maxwt = $\sum_{i=1}^{21} TC[i]$ and Minwt = TC[19]. If all the *TCN* values in the *TC* list equal 1, then Maxwt = 19 and Minwt = 1, therefore wt = 10.

As we can see, the amount of time saving increases with the increase of the number of nodes. In Diagram 12, when the number of nodes increases from 10 to 400, the time saving increases from 13205 to 22125, 54050, 286065,…, up until 12225450 time units.

Next, we check the average time saved in *each node* after Proposition 1 is used once to find the correct result of a request. The following diagram shows the experimental results achieved:

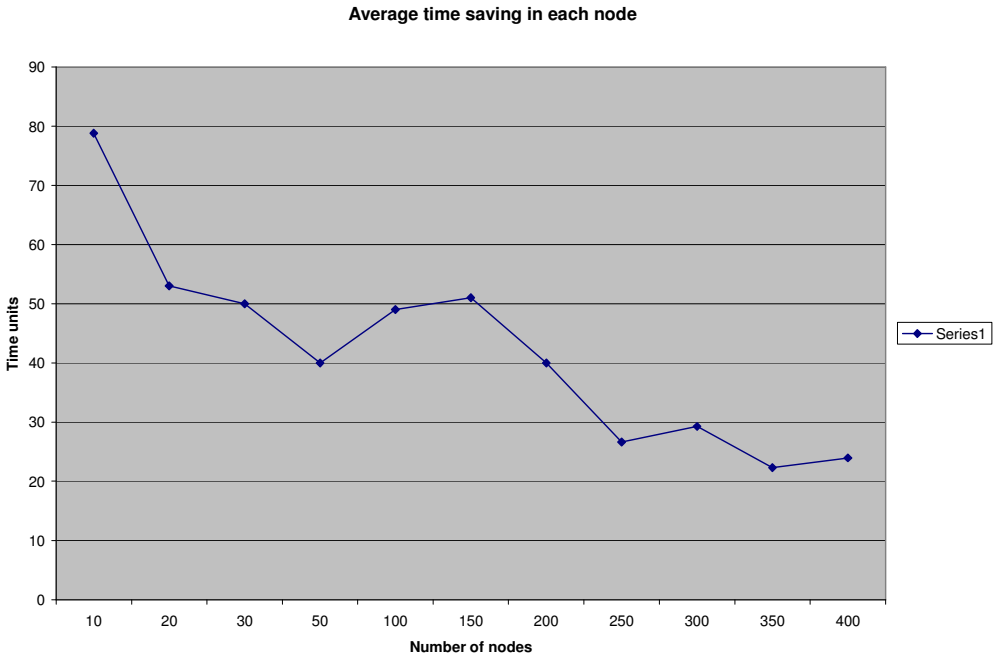

Figure 13 Average time saving in each node

This diagram shows that, although the total time saving increases with the increase of the number of nodes, the time saving *in each node* achieved by a one-time execution of Proposition 1 is reduced. The increase in total time saving is not caused by the increase of the amount of time saving in each node, but caused by the increase of the number of nodes which contribute to the accumulation of the total time saving. Furthermore, as mentioned before in Section 5.1.1: evaluation when TN changes, when the number of nodes increases while the percentage of faulty nodes is fixed, the time for the voter to find the correct result of a request *q* is delayed. This causes the execution of Proposition 1 invoked by the verdict of *q* to be delayed as well. This delay can have two side effects for each request *q'* whose correct result can be found by the execution of this proposition:

- The number of nodes calculated *q'* increases, therefore reducing the time saving
- The index (*indq'*) of *q'* in W list decreases, therefore reducing the time saving.

Evaluation when FN changes

The following diagram shows the *total* average time saved in reaching the correct results of the requests while the *percentage of faulty nodes changes*. In this evaluation we have kept the values of the parameters TN and RT fixed while the value of FN changes. In the following diagram, TN = 200, RT = 0, each node sends 15 distinct request throughout the simulation run, and FN ranges from 5% to 49% of the total nodes. Each node becomes faulty with a probability of 50%. Since each node can become faulty at anytime, and nodes can send out requests with any random pace, each data in the following diagram is the average of 100 simulation runs.

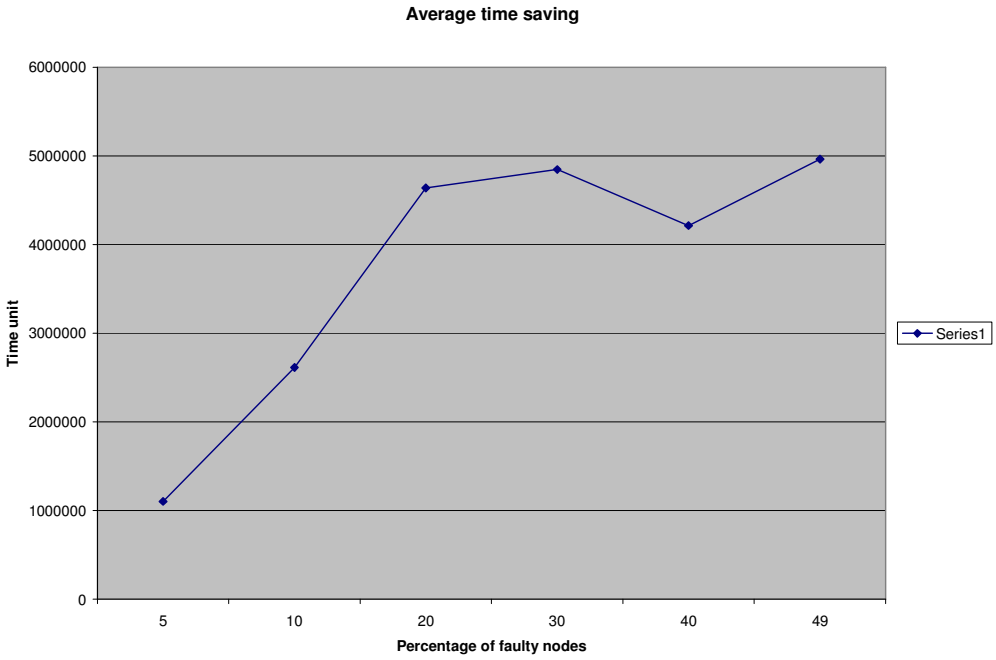

Figure 14 Average time saving

As it can be seen, with the increase in percentage of faulty nodes, the total time saved (accumulated from all nodes) to reach a correct result increases. Each data in the presented diagram represents the value of *wt*. The increase can be explained with the fact that, with the increase of the number of faulty nodes, the time needed for the voter to find the correct result for a request increases as well. With the increase of faulty nodes, more time is needed for a voter to find a correct result. Therefore, when Proposition 1 finds the correct result for a request, more time is saved by the voter. For example, consider two systems with the same number of nodes (ex. 200), one with 10% faulty nodes, and the

other with 30% faulty nodes. In both systems a situation is reached where Proposition 1 finds the correct result *r* for request *q*, while the voter has only received 10% of the total number of results it needs to declare *r* as the correct value. In this situation the time saving in the first system is:

> Number of common results the voter needs to make a decision: 10% of 200 nodes = 20,
> 10% of 20 result has been received by the voter = 2
> Number of results still needed by the voter: 20-2+1 = 19
> The time saving is: Maxwt = $\sum_{i=1}^{21} TC[i]$ and Minwt = TC[19]. In the worst case, all the *TCN* values in the *TC* list equal 1, then Maxwt = 19 and Minwt = 1, therefore wt = 10.

On the other hand the time saving for the system with 30% faulty node is:

> Number of common results the voter needs to make a decision: 30% of 200 nodes = 60,
> 10% of 60 result has been received by the voter = 6
> Number of results still needed by the voter: 60-6+1 = 55
> The time saving is: Maxwt = $\sum_{i=1}^{55} TC[i]$ and Minwt = TC[55]. If all the *TCN* values in the *TC* list equal 1, then Maxwt = 55 and Minwt = 1, therefore wt = 28.

As we can see, the amount of time saved increases with the increase of the number of faulty nodes. This is confirmed in Diagram 15: when the percentage of faulty nodes increases from 5% to 49%, the time saved increases from 1102385 to 4963330 time units.

In addition to the *total time saved*, we check to see the average time saved *in each node* after a Proposition 1 finds the correct result of a request. The following diagram shows the results achieved:

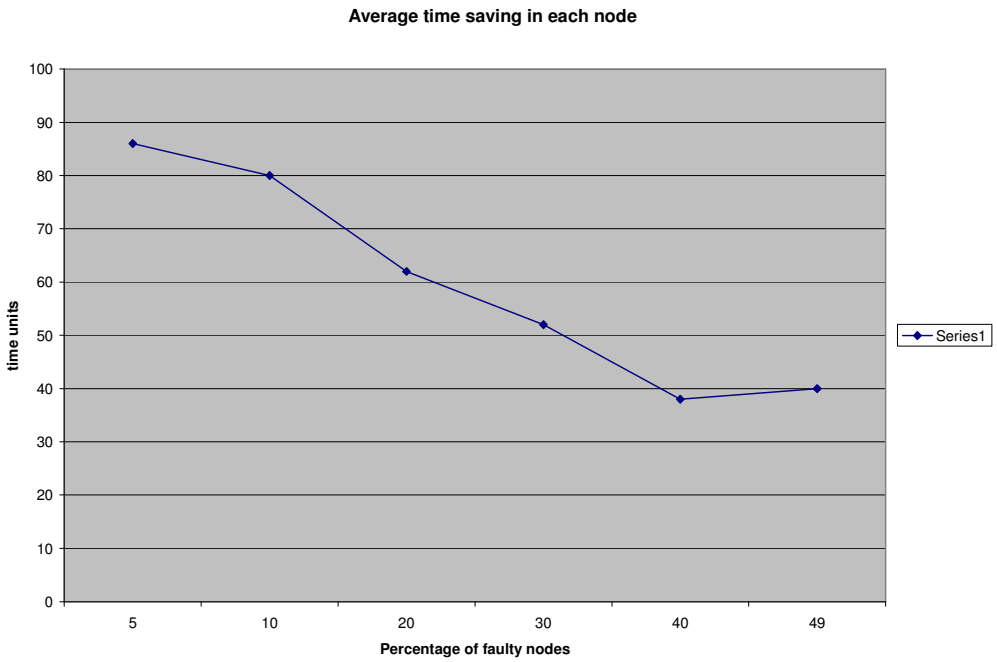

Figure 15 Average time saving in each node

This diagram shows that although the total time saving increases with the increase of the number of faulty nodes, the time saving in each node achieved by a one-time execution of Proposition 1 is reduced. This can be explained as follows: when the number of faulty nodes increases, the accumulated time saving of all the nodes increases as well; this is caused by the *increase of the number of nodes* where time saving can be achieved. However, the amount of time saved in *each node* does not increase. Consider the above example. This time, suppose that all the *TCN*'s in the *TC* list of the (second) system with 30% faulty nodes have a value of 0.5. In this case, its (total) time saving is: wt = 14. As we can see, although the *total* average time saving of the second system is more than the total average time saving of the first system with 10% faulty nodes where wt = 10 and TCN of each node is 1, the time saved in *each* node is smaller in the second system compared to that of the first system.

Evaluation when RT changes

The following diagram shows the total average time saved in reaching the correct results of the requests while the number of repeated requests changes. In this evaluation we have

kept the values of the parameters TN and FN fixed while the value of RT changes. In the following diagram TN = 200, FN = 49% while each node sends 15 requests throughout the simulation run. TN ranges from 5 to 15. Since each node can become faulty at anytime, and nodes can send out requests with any random pace, each data in the following diagram is the average of 100 simulation runs.

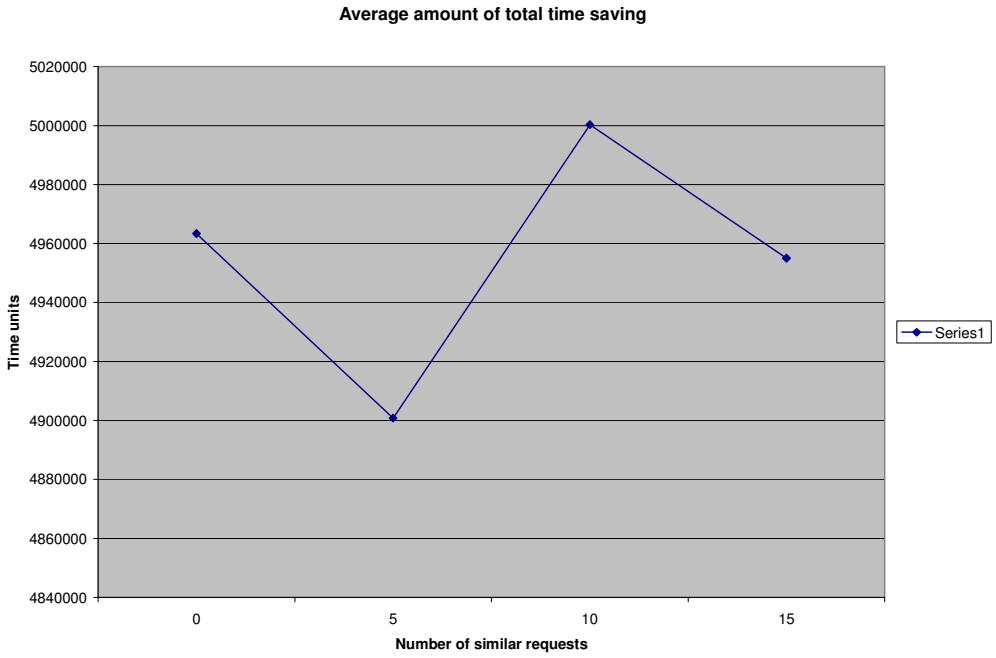

Figure 16 Average amount of total time saving

As it can be seen the total amount of time saving does not change significantly. It has an approximate value of 4940000 time units. The reason the time saving does not change is that the increase in the number of a repeated request does not speed up the time it takes for its correct result to be found. When the voter receives a result of a request *q* from a node *n*, it checks if *n* has previously sent in a result for *q*. If it hasn't, the voter adds the new result to its Q list and considers it when finding the correct result for *q*; otherwise the voter will discard it. Thus, the process of finding the correct result of a repeated request is similar to the situation where each request is sent out only once.

5.2 Evaluating the tradeoff

While gaining savings on the total number of redundant computations and reducing the time to reach correct results, the proposed method introduces additional communications between the voting center and the other nodes. After having reached a verdict for a requests *q*, the voting center needs to send additional messages to all other nodes in order to enable them to remove request *q* from their W list, and to find correct results to requests in their C list by using Proposition 1. For example, suppose each service request is sent to k nodes.

- The voting center uses k additional messages to send a verdict *r* to all k nodes. In this case, there are altogether k+1 additional messages passed around so that all the nodes can find the correct result of request q.
- In addition, when a node finds the correct result of a request *q'* using Proposition 1, the node sends the correct result to the voter and the voter forwards this result to all the other *k-1* remaining nodes. In this case, for all the nodes to find the correct result of request *q'*, *k* additional messages are passed around.

To better understand the trade-off, we have carried out experiments to show the relationship between the numbers of requests whose computation is exempted and the number of additional messages introduced. The following diagram shows the ratio of *numbers of redundant computations avoided / the number of additional messages*. To estimate this ratio we have considered a weight for each message and each request based on the amount of time they consume.

In our evaluation, we consider a weight with a value of 1 for each message and a weight with a value of 5 for the processing of each request. Note that we are evaluating our method in a computation intensive environment, which is why request processing has been given a larger weight. Therefore the ratio is computed as follows:

$$\frac{5 * number\ of\ redundant\ computations\ avoided}{1 * number\ of\ additional\ messages}$$

Diagram 17 shows the ratio when FN = 49%, RT=0 while each node sends 15 distinct

requests. TN varies from 10 to 400 nodes. Again, due to the existence of other factors, each data in the figure is calculated as an average from 100 experiments.

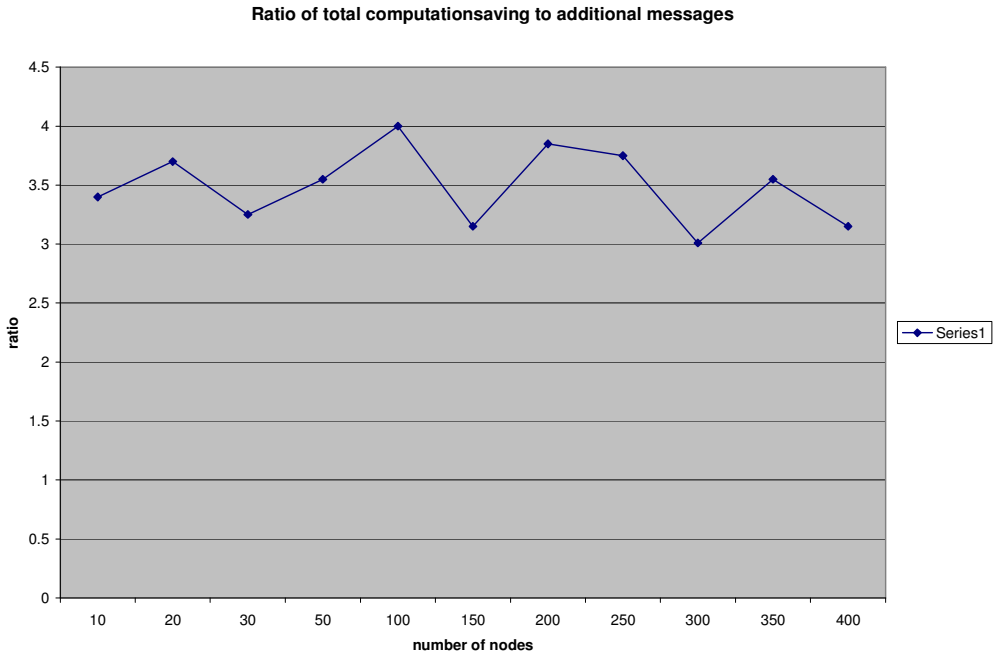

Figure 17 Ratio of total computation saving to additional messages

This diagram shows that although our method adds new messages into the system, it can achieve computation saving with an average ratio of 3.5. In addition, the following diagram shows the ratio of the time reduced to reach the correct result of a request to the number of additional messages introduced. As mentioned we have assumed that one time unit is consumed for a single message to arrive at its destination.

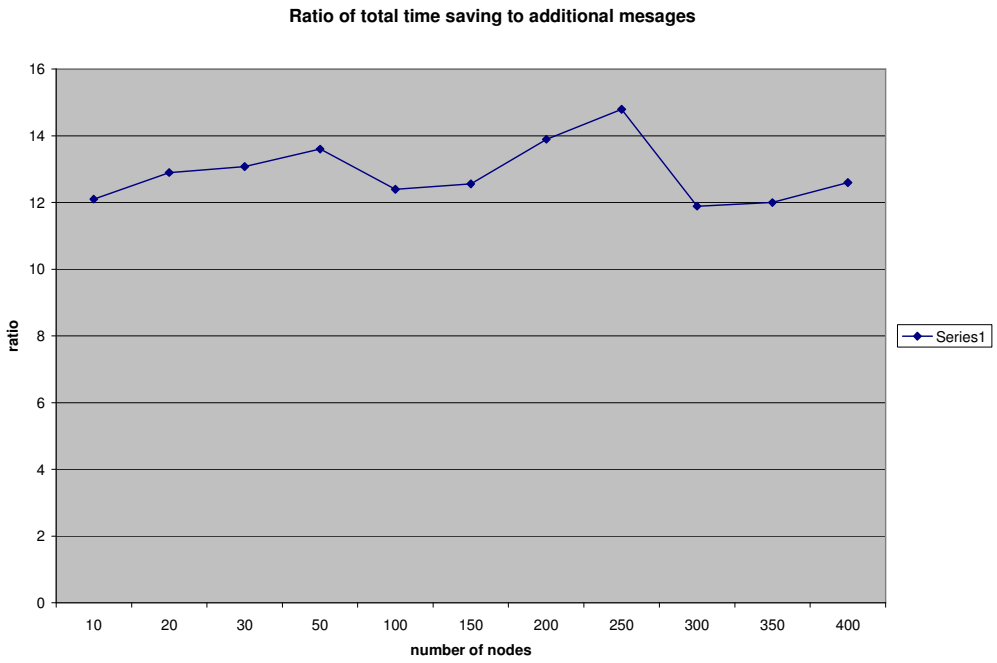

Figure 18 Ratio of total saving to additional messages

As we can see, the ratio of the *time reduced to reach the correct result* / *the number of additional messages* remains more or less a constant (approximately 12).

Chapter 6. Conclusion and Future Work

The work presented in this thesis mainly focused on developing an efficient fault tolerant method for distributed systems, in terms of the amount of time consumed and number of computations carried out to find the correct result of a request.

The fact that the results produced by a node can help us understand the underlying behavior of a node helped us establish an efficient fault tolerant method. As shown, the effect of different behaviors (correct or incorrect) of a node can exceed more than one request. Therefore, by finding the range and type of the effect of different behaviors of a node, the correctness or incorrectness of the results of multiple requests can be found simultaneously. As a consequence, by using this information we were able to reduce the time to find the correctness of requests.

In addition, we used the dynamicity of the system requirements to reduce computations. The computations needed to be executed so the correct result of a request can be found changes as different nodes carry out different requests. Therefore, if the lists of requests to be computed by the nodes are compatible with the system requirements, unnecessary computations can be avoided, resulting in the reduction of computations carried out.

This thesis showed that by using the nodes to aid the voter in finding the correct results, efficiency can be achieved in terms of time complexity and number of computations. We presented two proposition based on our following observations:

1. when a node completes a computation incorrectly, we can conclude, without further computation, that all the results of the computations after this point (until its latest recovery) are incorrect.
2. when a node completes a computation correctly, we can conclude, without further computation, that all the results of the computations before this point (after its most recent recovery) are correct.

Our work is one of the first to make use of the two observations above to develop an efficient fault tolerant method. In our method, when a node receives a correct result for a previously calculated request from the voter, we use the above two observations to conclude that the results the node has calculated for previous requests is correct or incorrect. Therefore, our method can find the correct result of a request before the central voter can find it, resulting in reduction of the time and number of computations needed to find the correct results to requests. Consequently, in our model when a correct result of a request is found the remaining nodes which have not yet calculated it are informed of it and are told not to calculate the request any longer. This results in an additional reduction of the number of computations executed by the nodes.

Our method is capable of achieving an average time saving of 5350000 time units when finding the correct result of a request while the number of active nodes increased from 10 nodes to 400 nodes. While this same method achieves an average time saving of 2300000 when the percentage of faulty nodes increase from 5% to 49%, and an average time saving of 4940000 when the redundant sends of a single request increases from 0 to 15.

In addition, by using the above Propositions our method is capable of achieving an average computation saving of 190657 computations while the number of nodes increases from 10 to 400. As well, this method achieves an average computation saving of 340133 computations when the percentage of faulty nodes increases from 5% to 49% and an average computation saving of 895000 computations when the redundant sends of a single request increases from 0 to 15.

Our method has a message complexity of $O(n^2)$ where n is the number of nodes. Although our method introduces additional messages into the system, its complexity is exponential in terms of the number of nodes, which is similar to the other commonly used fault tolerant methods [7, 8, 12, 15].

In addition the algorithm which makes use of Proposition 1 and 2, which is the heart of our method, has a time complexity of $O(nq^2)$ for a request q, where *nq* is the total number of requests.

As it was seen, our method successfully reduces the number of computations and time consumption to find the correct result or a request, therefore developing an efficient fault tolerant method for distributed systems.

As for our future work, we plan to advance our method to stateful distributed fault tolerant systems. At the moment our method can not handle the diversion in results caused by different ordering of requests in different nodes. To be able to advance our method to stateful systems, we aim at using Proposition 1 and 2 therefore we do not want to achieve similar ordering of requests in all the nodes. On the other hand we aim at finding the correct results despite the diversions in results from correct nodes and advancing Proposition 1 and 2 so nodes can be able to find correct and incorrect results despite being in relatively different states from each other.

In addition, we aim at distributing the voting strategy among the nodes so our method will not suffer a single point of failure in terms of the voting center.

References

[1] M. AbdElMalek, G. R. Ganger, G. R. Goodson, M. K. Reiter, J. J. Wylie, "Fault-Scalable Byzantine Fault Tolerant Services", *In Proceedings of the 20th ACM Symposium on Operating Systems Principles*, Publisher: ACM, pages 59-74, 2005.

[2] A. S. Aiyer, L. Alvisi, A. Clement, M. Dahlin, J. P. Matrin, C. Porth, "BAR Fault Tolerance for Cooperative Services", *ACM SIGOPS Operating Systems Review*, Publisher: ACM, vol. 39, issue 5, pages 45-58, 2005.

[3] E. A. P. Alchieri, A. N. Bessani, J. da Silva Fraga, "A Dependable Infrastructure for Cooperative Web Services Coordination", *Proceedings of the 2008 IEEE International Conference on Web Services*, Publisher: IEEE Computer Society, pages 21-28, 2008.

[4] Y. Amir, B. Coan, J. Kirsch, J. Lane, "Customizable Fault Tolerance for Wide-Area Replication", *Proceedings of the 26th IEEE International Symposium on Reliable Distributed Systems*, Publisher: IEEE Computer Society, pages: 65-82, 2007.

[5] Y. Amir, C. Danilov, D. Dolev, J. Kirsch, J. Lane, C. Nita-Rotaru, J. Olsen, D. Zage, "Scaling Byzantine Fault-Tolerant Systems to Wide Area Networks**",** *Proceedings of the International Conference on Dependable Systems and Networks*, Publisher: IEEE Computer Society, pages 105-114, 2006.

[6] A. Bagchi, S. L. Hakimi, "An Optimal Algorithm For Distributed System Level Diagnosis", *Twenty-First International Symposium of Fault Tolerant Computing*, pages 214-221, 1991.

[7] M. Castro, B. Liskov, "Practical byzantine fault tolerance and proactive recovery", *ACM Transactions on Computer Systems*, Publisher: ACM, vol. 24, issue 4, pages 398-461, 2002.

[8] M, Correia, N. Ferreira Neves, P. Ver´ıssimo, "How to Tolerate Half Less One Byzantine Nodes in Practical Distributed Systems", *Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems*, Publisher: IEEE Computer Society, pages 174-183, 2004.

[9] E. P. Duarte Jr., T. Nanya, "A Hierarchical Adaptive Distributed System-Level Diagnosis Algorithm", *IEEE Transactions on Computers*, Publisher: IEEE Computer Society, vol. 47,issue 1, pages 34-45, 1998.

[10] E. P. Duarte, Jr. , A. Brawerman , L. Carlos, P. Albini, "An Algorithm for Distributed Hierarchical Diagnosis of Dynamic Fault and Repair Events", Proceedings of the Seventh International Conference on Parallel and Distributed Systems, Publisher: IEEE Computer Society, pages 299, 2000.

[11] P. Feldman, S. Micali, "Optimal algorithms for Byzantine agreement", *Proceedings of the twentieth annual ACM symposium on Theory of computing*, Publisher: ACM, pages 148-161, 1988.

[12] M. J. Fischer, "The Consensus Problem in Unreliable Distributed Systems (A Brief Survey)", *Proceedings of the 1983 International FCT-Conference on Fundamentals of Computation Theory*, Publisher: Springer-Verlag, pages 127-140, 1983.

[13] M. Fitzi, U. Maurer, "Efficient Byzantine Agreement Secure Against general Adversaries", *Proceedings of the 12th International Symposium on Distributed Computing*, Publisher: Springer-Verlag, pages 134-148, 1998.

[14] B. Hardekopf, K. Kwiat, and S. Upadhyaya. "Secure and fault-tolerant voting in distributed systems". *In IEEE Proceedings of Aerospace Conference*, Publisher: IEEE Computer Society, pages 1117-1126, 2001.

[15] S. H. Hosseini, J. G. Kuhl, S. M. Reddy, "On Self-Fault Diagnosis of the Distributed Systems", *IEEE Transactions on Computers*, Publisher: IEEE Computer Society, vol. 37, issue 2, pages 248-251, 1988.

[16] S. H. Hosseini, "On Fault-Tolerant Structure, Distributed Fault-Diagnosis, Reconfiguration, and Recovery of the Array Processors", *IEEE Transactions on Computers*, Publsiher: IEEE Computer Society, vol. 38, issue 7, pages 932-942, 1989.

[17] K. P. Kilstrom, L. E. Moser, P. M. Melliar-Smith, "Byzantine Fault Detectors for Solving Consensus", *The Computer Journal*, Publisher: Oxford University Press, vol. 46, issue 1, pages: 16-35, 2003.

[18] H. Kopetz, A. Damm, C. Koza, M. Mulazzani, W. Schwabl, C. Senft, R. Zainlinger, "Distributed Fault-Tolerant Real-Time Systems: The Mars Approach", *IEEE Micro*, vol. 9, issue 1, pages 25-40, 1989.

[19] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, E. Wong, "Zyzzyva: Speculative Byzantine Fault Tolerance", *ACM Symposium on Operating Systems Principles*, Publisher: ACM, pages 45-58, 2007.

[20] L. Lamport, R. Shostak, M. Pease, "The Byzantine Generals Problem", *ACM Transactions on Programming Languages and Systems (TOPLAS)*, Publisher: ACM, Vol. 4, issue 3, pages 382-401, 1982.

[21] G. Latif-Shabgahi, J. M. Bass, S. Bennett, "A taxonomy for software voting algorithms used in safety-critical systems", IEEE Transaction of Reliability, Publisher: IEEE Computer Society, vol. 53, issue 3, pages 319-328, 2004.

[22] G. Latif-Shabgahi, S. Bennett,Adaptive, "Integrating selected fault masking and self-diagnosis mechanisms", *Euromicro Workshop on Parallel and Distributed Processing*, pages 97-104, 1999.

[23] N. Looker, M. Munro, J. Xu, "Increasing Web Service Dependability Through Consensus Voting", *Proceedings of the 29th Annual International Computer Software and Applications Conference*, Publisher: IEEE Computer Society, pages 66-69, 2005.

[24] N. A. Lynch, M. J. Fischer, R. J. Fowler, "A Simple and Efficient Byzantine Generals Problem", *Tech. Rep. GIT-ICS-82/02, School of Information and Computer Science, Georgia Institute of Technology*, Atlanta, Georgia, 1982.

[25] M. G. Merideth, A. Iyengar, T. Mikalsen, S. Tai, I. Rouvellou, P. Narasimhan, "Thema: Byzantine-Fault-Tolerant Middleware for Web-Service Applications", *24th IEEE Symposium on Reliable Distributed Systems*, pages 131-140, 2005.

[26] S. Rangarajan, A. T. Dahbura, E. A. Ziegler, "A Distributed System-Level Diagnosis Algorithm for Arbitrary Network Topologies", IEEE *Transactions on Computers*, Publisher: IEEE Computer Society, vol 44, issue 2, pages 312-334, 1995.

[27] R. Rodrigues, M. Castro, B. Liskov, "BASE: Using Abstraction to Improve Fault Tolerance", *ACM Transactions on Computer Systems*, Publisher: ACM, vol. 21, issue 3, pages 236-269, 2003.

[28] G. T. Santos, L. C. Lung, C. Montez, **"**FTWeb: A Fault Tolerant Infrastructure for Web Services", *Proceedings of the Ninth IEEE International EDOC Enterprise Computing Conference*, Publisher: IEEE Computer Society, pages 95-105, 2005.

[29] P. Stelling, C. DeMatteis, I. Foster, C. Kesselman, C. Lee and G. von Laszewski, "A fault detection service for wide area distributed computations", *Proceedings of the 7th IEEE International Symposium on High Performance Distributed Computing*, Publisher: IEEE Computer Society, pages 268, 1998.

[30] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: a tutorial", *ACM Computing Surveys*, Publisher: ACM, vol. 22, issue 4, pages 299-319, 1990.

[31] P. Sobe, "Reaching Efficient Fault-Tolerance for Cooperative Applications", *IEEE International Proceedings of Computer Performance and Dependability Symposium*, Publisher: IEEE Computer Society, pages 48-57, 2000.

[32] A. K. Somani, N. H. Vaidya, "Understanding Fault Tolerance and Reliability", *Computer*, Publisher: IEEE Computer Society Press, vol. 30, issue 4, pages 45-50, 1997.

[33] J. Yin, J. P. Martin, A. Venkataramani, L. Alvisi, M. Dahlin, "Separating Agreement from Execution for Byzantine Fault Tolerant Services", *ACM Symposium on Operating Systems Principles*, Publisher: ACM, pages 253-267, 2003.

[34] W. Zhao, "A Lightweight Fault Tolerance Framework for Web Services", *IEEE/WIC/ACM International Conference on Web Intelligence*, Publisher: IEEE Computer Society, pages 542-548, 2007.

[35] W. Zhao, "Byzantine Fault Tolerance for Nondeterministic Applications", *Third IEEE International Symposium on Dependable, Autonomic and Secure Computing*, pages 108-118, 2007.

Vita Auctoris

Samaneh Navabpour was born in 1983 in Tehran, Iran. She was able to top the university entrance exam and be granted entry to one of the top four universities of Iran, Shahid Beheshti University. After achieving her B. E in Software Engineering in 2005, for two years she worked in two of the most prestigious research centers in Iran, Institute for Studies in Theoretical Physics and Mathematics (IPM) and Niroo Research Institute (NRI). Currently she is a candidate for the Masters Degree in Computer Science at the University of Windsor and is planning to graduate in Summer 2009.