The Weak Snapshot Abstraction

Bachelor’s Thesis

Andrina Arnold
anarnold@ethz.ch

Distributed Computing Group
Computer Engineering and Networks Laboratory
ETH Zürich

Supervisors:
Mr. Wang Ye
Prof. Dr. Roger Wattenhofer

September 7, 2021
Acknowledgements

I would like to thank Mr. Wang Ye for supervising this thesis and for providing regular feedback, guidance, and insights about the topic.
Abstract

With increasing globalization, a more disruptive trade and political environment, and an increased risk of natural disasters due to climate change, many companies have spread thousands or even millions of machines across all continents in order to protect computer and information systems as well as to store data more reliably. Due to this increase in size and complexity of distributed systems, communication between processes is no longer limited to point-to-point communication protocols. To facilitate communication between multiple processes in a distributed system without a consensus guarantee, reliable broadcast abstractions play a central role. To enable an asynchronous reliable broadcast in a dynamic byzantine message passing system where consensus is not guaranteed, an asynchronous dynamic reliable byzantine broadcast algorithm needs to be developed that provides a mechanism called reconfiguration operation to ensure dynamism.

In this thesis, we designed a weak snapshot abstraction that can be used as an underlying building block to implement reconfiguration operations in an asynchronous dynamic byzantine message passing system where consensus is not guaranteed, e.g. in an asynchronous dynamic reliable byzantine broadcast algorithm.
# Contents

Acknowledgements                                  i  
Abstract                                           ii  

1 Introduction                                    1  
1.1 Motivation                                    1  
1.2 Objectives                                    2  
1.3 Related Work                                  2  

2 System Model and Specification                  3  

3 The Weak Snapshot Abstraction                   4  
3.1 Asynchronous Reliable Broadcast Algorithm     4  
3.2 The Weak Snapshot Algorithm                   5  
3.2.1 Semantics of the $update_i(c)$ and $scan_i()$ Operation   5  
3.2.2 Implementation                             5  
3.2.3 collect() Procedure                        5  
3.2.4 $update_i(c)$ Operation                    6  
3.2.5 $scan_i()$ Operation                       7  
3.2.6 Proofs for Property of Mem Array and Properties NV1-NV5 10  

4 Conclusion                                     17  
4.1 Conclusion                                   17  
4.2 Future Work                                  17  

Bibliography                                     19
1.1 Motivation

Although many networks already offer reliable communication channels such as the underlying, itself unreliable Internet Protocol (IP) in combination with Transmission Control Protocol (TCP), these networks still do not offer sufficient reliability for many applications.[1] Operating in a distributed system some senders may only know and see a strict subset of processes in the system. Thus, a message sent by such senders will only be reached by a strict subset of processes in the same system.

Deploying an asynchronous reliable broadcast algorithm, a process can send a message to several other processes simultaneously and if a correct process accepts the message, then the message will eventually be accepted by every correct process of the system.[2] Consequently, every correct process will eventually accept the message. Asynchronous reliable broadcast algorithms play a central role because they do not require a consensus guarantee. To assure that asynchronous reliable broadcast algorithms work safely and up to expectations, asynchronous reliable broadcast algorithms must be fault-tolerant. This is because a distributed system may contain byzantine processes that can behave in a way that could destroy the system.

As some of the processes in the system become slow and outdated or have even died, one wants to replace them with new, fast ones. Therefore, asynchronous reliable broadcast algorithms need to provide a reconfiguration operation that supports the adding and removing of processes to and from a system. A distributed system that fulfills these requirements is dynamic. Providing dynamism results in stable, reliable, durable, and long-lasting distributed systems.

An asynchronous dynamic reliable byzantine broadcast algorithm is the solution to meet all the above mentioned requirements. Asynchronous dynamic reliable byzantine broadcast algorithms have become more and more popular in many applications such as in cryptography, e.g. to implement threshold secret sharing in an asynchronous dynamic byzantine message passing systems without
1. Introduction

There is no consensus guarantee.

1.2 Objectives

The main goal of this thesis is to develop a weak snapshot abstraction that can be used as an underlying building block to implement reconfiguration operations in asynchronous dynamic byzantine message passing systems where consensus is not guaranteed, e.g., in an asynchronous dynamic reliable byzantine broadcast algorithm.

Since the weak snapshot abstraction is used in algorithms where consensus is not guaranteed, the sequence of configurations for each process in the weak snapshot abstraction differs. Therefore, the property that it is sufficient that a sequence of configuration exists must be enabled by a weak snapshot abstraction. For processes in a weak snapshot abstraction, knowing exactly which configurations belong to that sequence is not necessary. The only requirement is that the processes have some assessment that includes these configurations. Other configurations that are not in the sequence may also be included by the assessment.[3]

For a weak snapshot object S, we are going to define a \textit{scan}_i() operation and an \textit{update}_i(c) operation on a given set of processes P. Additionally, we must ensure that all scan operations that see any updates must see the "first" update.

1.3 Related Work

We extract and adapt some of the ideas and concepts defined in the work [3], which solves the atomic R/W storage problem in a dynamic setting without consensus or stronger primitives. In this work, a weak snapshot abstraction was specified that operates in a distributed system and was implemented based on a collection of processes that interact utilizing asynchronous message passing. The weak snapshot abstraction described in this work was based on a system that only allows processes to crash but does not permit processes to be byzantine.

In the work [2], an asynchronous reliable broadcast algorithm was designed to work in a distributed system that might contain up to f byzantine processes out of n=3f+1 processes. Their asynchronous reliable broadcast algorithm was based on a static distributed system which means that the system does not permit processes to be added or removed.
The weak snapshot abstraction is deployed on a distributed system and it is implemented based on a collection of processes that interact utilizing asynchronous message passing. We assume a universe of processes $\Pi$ to be unknown and unbounded, possibly infinite. Furthermore, it is a precondition that the communication channels between the processes are reliable.\cite{3}

A set of processes $P \subseteq \Pi$ can access a weak snapshot object $S$.\cite{3} We assume that a process $p_i \in P$ knows about all processes $p_j \in P$. We denoted the number of all processes $p_j \in P$ by $n_P$. We assume that at most $f_P$ byzantine nodes are contained in $P$ where the size of the set $P$ is $n_P = 4f_P + 1$.

For each process $p_i \in P$ we define an array $\text{Mem}$ of local atomic registers of length $n_P$. All local atomic registers are initialized to $\bot$. We must ensure that each register $\text{Mem}[i]$ contains at most one value which is not $\bot$. This is proven in section 3.2.6 under the headline Property of Mem Array.
A set of processes $P$ can access a weak snapshot object $S$. For each process $p_i \in P$ we define two operations, namely $update_i(c)$ and $scan_i()$. Essential is that $p_i$ first terminates all previous outstanding operations on the weak snapshot before $p_i$ invokes the new operation on the weak snapshot.[3]

Both the $update_i(c)$ and the $scan_i()$ operation use the procedure collect() in their implementation.

The $update_i(c)$ operation first checks if another $update_j(c')$ operation was previously successful by using the collect() procedure. If this is not the case, the $update_i(c)$ operation will take a value $c$ as an input and will utilize a slightly modified version of the asynchronous reliable broadcast algorithm [2] to do the update. Finally, process $p_i$ sends $\langle \text{OK}, update_i(c) \rangle$ to all $p_k \in P$ regardless of what the collect() procedure returned to indicate that the $update_i(c)$ operation has successfully completed.

The $scan_i()$ operation uses the procedure collect() to return a set of values previously written by an $update_j(c')$ operation.

### 3.1 Asynchronous Reliable Broadcast Algorithm

The asynchronous reliable broadcast algorithm [2] satisfies the following properties:

1. If a correct node broadcasts a message reliably, it will eventually be accepted by every other correct node.[2]

2. If a correct node has not broadcast a message, it will not be accepted by any other correct node.[2]

3. If a correct node accepts a message, it will be eventually accepted by every correct node.[2]
3. The Weak Snapshot Abstraction

3.2 The Weak Snapshot Algorithm

3.2.1 Semantics of the $update_i(c)$ and $scan_i()$ Operation

"We require the following semantics for the $update_i(c)$ and $scan_i()$ operation:" [3]

NV1 "Let $o$ be a $scan_i()$ operation that returns $C$. Then for each $c \in C$, an $update_j(c)$ operation is invoked by some process $p_j$ prior to the completion of $o$." [3]

NV2 "Let $o$ be an $scan_i()$ operation that is invoked after the completion of an $update_i(c)$ operation, and that returns $C$. Then $C \neq \emptyset$." [3]

NV3 "Let $o$ be a $scan_i()$ operation that returns $C$ and let $o'$ be a $scan_i()$ operation that returns $C'$ and is invoked after the completion of $o$. Then $C \subseteq C'$." [3]

NV4 "There exists a $c$ such that for every $scan_i()$ operation that returns $C' \neq \emptyset$, it holds that $c \in C'$." [3]

NV5 "If some majority $M$ of processes in $P$ keep taking steps then every $scan_i()$ and $update_i(c)$ invoked by every $p_i \in M$ eventually completes." [3]

3.2.2 Implementation

The set $C_i$ returned by a $scan_i()$ operation does not have to contain the value of the most recently completed update that precedes it. Thus, it holds that in the weak snapshot algorithm we do not demand all updates to be ordered as in atomic snapshot objects. This means that all scans that see any updates see the "first" update.[3]

Let $scan_i()$ operation $o$ be the first to complete its first collect(). This implies that any other $scan_i()$ operation $o'$ starts its second collect() only after $o$ completes its first collect().[3]

3.2.3 collect() Procedure

First, $p_i$ initialize $C_i$ and ReplyCollect to the empty set and sends <COLLECT> to all $p_k \in P$. Each process $p_k \in P$ receiving <COLLECT> from $p_j$ sends <REPLY COLLECT, Mem, $p_k$> to $p_j$. $p_i$ waits for <REPLY COLLECT, Mem, $p_k$> from $3fp + 1$ processes $p_k \in P$ because there are at most $fp$ byzantine processes. Each <REPLY COLLECT, Mem, $p_k$> message received by $p_i$ is added to the set ReplyCollect. $p_i$ iterates with $j$ over the whole Mem array and checks if there
are at least $f_P + 1$ processes $P_k$ which have the same value $c$ at Mem[j].Read(). If this the case, $p_i$ adds $c$ to the set $C_i$.

$p_i$ waits for $<\text{OK RECEIVED}, \text{update}_j(c), p_q>$ from $f_P + 1$ processes $p_q \in P$, to ensure that the $\text{update}_j(c)$ operation completes before the collect() procedure completes. After an $\text{update}_j(c)$ operation completes, $p_j$ will send $<\text{OK}, \text{update}_j(c)>$ to all $p_k \in P$. Therefore, each process $p_k$ which receives $<\text{OK}, \text{update}_j(c)>$ from $p_j$ knows that the $\text{update}_j(c)$ operation has successfully completed. Before sending this information to all processes $p_q \in P$, $p_k$ waits for $<\text{WRITTEN}(p_q, \text{update}_j(c))>$ from $3f_P + 1$ processes $p_q$. This ensures that byzantine nodes could not directly send $<\text{OK RECEIVED}, \text{update}_j(c), p_k>$ messages for any $p_k \in P$ to all $p_q \in P$ to make $p_i$ believe that the $\text{update}_j(c)$ operation has successfully completed. After $p_k$ has received $<\text{WRITTEN}(p_q, \text{update}_j(c))>$ from $3f_P + 1$ processes $p_q \in P$, $p_k$ sends $<\text{OK RECEIVED}, \text{update}_j(c), p_k>$ to all $p_i \in P$.

In the end, the collect() procedure returns the set $C_i$.

### 3.2.4 \text{update}_i(c)$ Operation

For the $\text{update}_i(c)$ operation, we use the asynchronous reliable broadcast algorithm [2] as a basis. If a process $p_i \in P$ wants to do an $\text{update}_i(c)$ operation, $p_i$ first invokes a collect().

If collect() returns a non-empty set, then some $\text{update}_j(c')$ operation has been successful and $p_i$ sends directly $<\text{OK}, \text{update}_i(c)>$ to all $p_k \in P$ indicating that the $\text{update}_i(c)$ has successfully completed.

If collect() returns an empty set, then no $\text{update}_j(c')$ operation has been successful before. Therefore, $p_i$ will send $<\text{REQUEST}(\text{update}_i(c))>$ to all $p_k \in P$. Process $p_i$ sends an $<\text{ECHO}(p_i, \text{request}(\text{update}_i(c)))>$ to all $p_k \in P$ because $p_i$ sends the $<\text{REQUEST}(\text{update}_i(c))>$ to all $p_k \in P$ and therefore $p_i$ has already received $<\text{REQUEST}(\text{update}_i(c))>$. On condition that a process $p_q \in P$ has received $<\text{ECHO}(p_k, \text{request}(\text{update}_j(c)))>$ from $f_P + 1$ processes $p_k \in P$ for the first time or $<\text{REQUEST}(\text{update}_j(c))>$ from $p_j$ for the first time, $p_q$ will send an $<\text{ECHO}(p_q, \text{request}(\text{update}_j(c)))>$ to all $p_k \in P$. It is necessary to only send an $<\text{ECHO}(p_q, \text{request}(\text{update}_j(c)))>$ for the first time a process $p_q \in P$ has received $<\text{ECHO}(p_k, \text{request}(\text{update}_j(c)))>$ from $f_P + 1$ processes $p_k \in P$ or $<\text{REQUEST}(\text{update}_j(c))>$ from $p_j$ because otherwise a byzantine node $p_j$ could send $<\text{REQUEST}(\text{update}_j(c))>$ and $<\text{REQUEST}(\text{update}_j(c))>$ and both request messages will be accepted.

After a process $p_q \in P$ has received $<\text{ECHO}(p_k, \text{request}(\text{update}_j(c)))>$ from $3f_P + 1$ processes $p_k \in P$, then $p_q$ accepts $<\text{REQUEST}(\text{update}_j(c))>$, writes the value $c$ to its local memory array Mem with Mem[j].Write(c) and sends $<\text{WRITTEN}(p_q, \text{update}_j(c))>$ to all $p_k \in P$. 

Process $p_i$ waits for $<\text{ECHO}(p_k, \text{request}(\text{update}_i(c)))>$ from $3f_P+1$ processes $p_k \in P$. Then, $p_i$ accepts the $<\text{REQUEST}(\text{update}_i(c))>$, writes the value $c$ to its local memory array Mem with Mem[$i$].Write($c$) and sends $<\text{WRITTEN}(p_i, \text{update}_i(c))>$ to all $p_k \in P$.

$p_i$ waits for $<\text{WRITTEN}(p_k, \text{update}_i(c))>$ from $3f_P+1$ processes $p_k \in P$. This ensures that at least $2f_P+1$ processes $p_k \in P$ have issued Mem[$i$].Write($c$). At most $f_P$ Byzantine nodes $p_f$ could send $<\text{WRITTEN}(p_f, \text{update}_i(c))>$ to all $p_q \in P$ without having issued Mem[$i$].Write($c$) or having written Mem[$i$].Write($c'$) with $c' \neq c$ and at most $f_P$ correct nodes $p_l \in P$ might not yet have received $<\text{ECHO}(p_k, \text{request}(\text{update}_i(c)))>$ from $3f_P+1$ processes $p_k \in P$ and thus, $p_l$ has not yet issued Mem[$i$].Write($c$) and has not yet sent $<\text{WRITTEN}(p_l, \text{update}_i(c))>$ to all $p_q \in P$.

After $p_i$ has received $<\text{WRITTEN}(p_k, \text{update}_i(c))>$ from $3f_P+1$ processes $p_k \in P$, $p_i$ sends $<\text{OK}, \text{update}_i(c)>$ to all $p_k \in P$. This waiting ensures that a Byzantine node could not just send $<\text{OK}, \text{update}_i(c)>$ to all $p_k \in P$ to make other processes think that the $\text{update}_i(c)$ operation has already successfully terminated.

### 3.2.5 $\text{scan}_i()$ Operation

First, we initialize $C$ to the set returned by the procedure $\text{collect}()$. If $C$ is the empty set, the operation $\text{scan}_i()$ directly returns the empty set. If $C$ is a non-empty set, we assign $C$ to the set returned by the second call of the procedure $\text{collect}()$. 


Algorithm 1 Weak Snapshot Algorithm - Code for Process $p_i$

1: procedure collect()
2: $C_i = \emptyset$
3: ReplyCollect = \emptyset
4: send $<\text{COLLECT}>$ to all $p_k \in P$
5: 
6: wait for $<\text{REPLY COLLECT}, \text{Mem}, p_k>$ from $3f_P+1$ processes $p_k \in P$
7: 
8: for received $<\text{REPLY COLLECT}, \text{Mem}, p_k>$ do
9: 
10: ReplyCollect $\leftarrow \{<\text{REPLY COLLECT}, \text{Mem}, p_k>\} \cup \text{ReplyCollect}$
11: end for
12: 
13: for $j \in \text{range}(n_P)$ do
14: for $<\text{REPLY COLLECT}, \text{Mem}, p_k> \in \text{ReplyCollect}$ do
15: 
16: 
17: end for
18: end for
19: 
20: return $C_i$
21: end procedure
22: 
23: upon receiving $<\text{OK, update}_j(c)>$ from $p_j$:
24: 
25: wait for $<\text{WRITTEN}(p_k, \text{update}_j(c))>$ from $3f_P+1$ processes $p_k$
26: send $<\text{OK RECEIVED, update}_j(c), p_q>$ to all $p_l \in P$
27: end upon
28: 
29: upon receiving receiving $<\text{COLLECT}>$ from $p_j$ by process $p_k \in P$:
30: 
31: send $<\text{REPLY COLLECT, Mem, p_k}>$ to $p_j$
32: end upon
3. The Weak Snapshot Abstraction

---

1: operation update\(_i\)(c)
2:     if collect() = ∅ then
3:         send <REQUEST(update\(_i\)(c))> to all \(p_k \in P\)
4:         send <ECHO(p_i, request(update\(_i\)(c)))> to all \(p_k \in P\)
5:     end if
6:     wait for <ECHO(p_k, request(update\(_i\)(c)))> from \(3f_P + 1\) processes
7:     accept REQUEST(update\(_i\)(c))
8:     Mem[i].Write(c)
9:     send <WRITTEN(p_i, update\(_i\)(c))> to all \(p_k \in P\)
10:    wait for <WRITTEN(p_k, update\(_i\)(c))> from \(3f_P + 1\) processes \(p_k\)
11:   end if
12:   send <OK, update\(_i\)(c)> to all \(p_k \in P\)
13: end operation
14: upon receiving <REQUEST(update\(_j\)(c))> from \(p_j\) for the first time or <ECHO(p_k, request(update\(_j\)(c)))> from \(f_P + 1\) processes \(p_k\) for the first time:
15:     send <ECHO(p_q, request(update\(_j\)(c)))> to all \(p_k \in P\)
16: end upon
17: upon receiving <ECHO(p_k, request(update\(_j\)(c)))> from \(3f_P + 1\) processes \(p_k\):
18:     accept <REQUEST(update\(_j\)(c))>
19:     Mem[j].Write(c)
20:     send <WRITTEN(p_q, update\(_j\)(c))> to all \(p_k \in P\)
21: end upon
22: operation scan\(_i\)()
23:     C ← collect()
24:     if C = ∅ then
25:         return ∅
26:     else
27:         C ← collect()
28:         return C
29:     end if
30: end operation
3.2.6 Proofs for Property of Mem Array and Properties NV1-NV5

**Property of Mem Array** For any \(p_i \in P\), the following holds:

a) if \(p_i\) receives \(<\text{WRITTEN}(p_q, \text{update}_i(c))>\) from \(3f_P+1\) processes \(p_q \in P\) and afterwards \(\text{Mem}[i]\).Read() from at least \(f_P+1\) processes \(p_k \in P\) returns \(c'\), then \(c'=c\).

b) if \(\text{Mem}[i]\).Read() from the \(f_P+1\) processes \(p_k \in P\) return \(c \neq \bot\) and \(\text{Mem}[i]\).Read() from the \(f_P+1\) processes \(p_j \in P\) return \(c' \neq \bot\), then \(c=c'\).

**Proof.** To ensure \(p_i\) receives \(<\text{WRITTEN}(p_k, \text{update}_i(c))>\) from \(3f_P+1\) processes \(p_k \in P\), any process \(p_i \in P\) has to accept \(\text{REQUEST}(\text{update}_i(c))\). We show that any \(p_i \in P\) accepts at most one \(\text{REQUEST}(\text{update}_i(c))\) in an execution.

To proof the lemma, we do a case split where we first (case 1) assume that \(p_i\) is a byzantine process and then (case 2) assume that \(p_i\) is a correct process.

First, we look at the case where \(p_i\) is a byzantine node. Suppose for the sake of contradiction that \(\text{REQUEST}(\text{update}_i(c))\) and \(\text{REQUEST}(\text{update}_i(c'))\) are accepted in the execution.

Each process \(p_q\) only sends \(<\text{ECHO}(p_q, \text{request}(\text{update}_i(c)))>\) to all \(p_k \in P\), if \(p_q\) receives \(<\text{REQUEST}(\text{update}_i(c))>\) from \(p_i\) for the first time or \(<\text{ECHO}(p_k, \text{request}(\text{update}_i(c)))>\) from \(f_P+1\) processes \(p_k \in P\) for the first time. This ensures that if \(p_j\) sends \(\text{REQUEST}(\text{update}_i(c))\) to some processes \(p_l \in Q\) with \(Q \subseteq P\) and \(\text{REQUEST}(\text{update}_i(c'))\) to some processes \(p_r \in R\) with \(R \subseteq P\), then at most one \(\text{REQUEST}(\text{update}_i(a))\) with \(a \in \{c, c'\}\) is accepted in an execution.

The reason for this is that any process \(p_k \in P\) would have to receive \(<\text{ECHO}(p_q, \text{request}(\text{update}_i(c)))>\) from \(3f_P+1\) processes \(p_q \in P\) to accept \(\text{REQUEST}(\text{update}_i(c))\) and \(<\text{ECHO}(p_i, \text{request}(\text{update}_i(c')))>\) from \(3f_P+1\) processes \(p_l \in P\) to accept \(\text{REQUEST}(\text{update}_i(c'))\). This is not possible because each correct process \(p_n\) can only send \(<\text{ECHO}(p_n, \text{request}(\text{update}_i(c)))>\) or \(<\text{ECHO}(p_n, \text{request}(\text{update}_i(c')))>\) and each byzantine process \(p_b\) can send both \(<\text{ECHO}(p_b, \text{request}(\text{update}_i(c)))>\) and \(<\text{ECHO}(p_b, \text{request}(\text{update}_i(c')))>\), which means that at most \(5f_P+1 < 6f_P+2\) \(\text{ECHO}(p_z, \text{request}(\text{update}_i(a)))\) for some \(a \in \{c, c'\}\) and any \(p_z \in P\) are sent. This contradicts our assumption that \(\text{REQUEST}(\text{update}_i(c))\) and \(\text{REQUEST}(\text{update}_i(c'))\) are accepted in the execution.

Now, we look at the case where \(p_i\) is a correct node. Suppose for the sake of contradiction that \(\text{REQUEST}(\text{update}_i(c))\) and \(\text{REQUEST}(\text{update}_i(c'))\) are accepted in the execution. We will observe the second accept in the execution which we define as the acceptance of \(\text{REQUEST}(\text{update}_i(c'))\).

In chapter 3 we state our assumption of a mechanism that always completes a previous operation on a weak snapshot object if any such operation has been
invoked and did not complete (because of restarts), whenever a new operation is invoked on the same weak snapshot object. Thus, when REQUEST(update\(_i\)(c')) is invoked, the update\(_i\)(c) has already completed and thus REQUEST(update\(_i\)(c)) is accepted.

Before we invoke REQUEST(update\(_i\)(c')), \(p_i\) completes collect(). By atomicity of the Mem array of each process \(p_k \in P\) and since the first REQUEST(update\(_i\)(c)) has successfully completed, \(p_i\) has received <WRITTEN\((p_k, \text{update}\(_i\)(c))\)> from at least at least \(3f+1\) \(p_k \in P\) indicating that at least \(2f+1\) processes \(p_q\) have issued Mem[i].Write(c) (because we have at most \(f\) byzantine nodes).

In the procedure collect(), \(p_i\) sends <COLLECT> to all \(p_n \in P\). At least \(3f+1\) processes \(p_k \in P\) will send <REPLY COLLECT, Mem, \(p_k\)> to \(p_i\) because there are at most \(f\) byzantine nodes. We iterate with \(j \in \text{range}(nP)\) over the whole array Mem and check for each \(j\), if there are at least \(f+1\) processes \(p_l\) which sent <REPLY COLLECT, Mem, \(p_l\)> to \(p_i\) and have Mem[i].Read()=c for any c.

Due to the fact that at least \(2f+1\) processes \(p_q\) have issued Mem[i].Write(c) in the update\(_i\)(c) operation and we wait for <REPLY COLLECT, Mem, \(p_k\)> from at least \(3f+1\) processes \(p_k\), it holds that at most \(f\) byzantine nodes and at most \(f\) correct nodes send <REPLY COLLECT, Mem, \(p_k\)> with a Mem array which does not contain Mem[i].Read()=c. This implies that at least \(f+1\) processes have Mem[i].Read()=c. Therefore, collect() returns a set containing the value c, and thus the condition collect()=\(\emptyset\) evaluates to FALSE.

This implies that we do not invoke REQUEST(update\(_i\)(c')) after the collect() completes returning FALSE and thus, we do not accept REQUEST(update\(_i\)(c')). This contradicts our assumption that REQUEST(update\(_i\)(c)) and REQUEST(update\(_i\)(c')) are accepted in the execution.

The proof of (a) follows directly from the fact that \(p_i \in P\) accepts at most one REQUEST(update\(_i\)(c)) for any c in an execution.

In order to proof (b), notice that if \(c \neq c'\), this means that both REQUEST(update\(_i\)(c)) and REQUEST(update\(_i\)(c')) are accepted in the execution, which contradicts the fact that any \(p_i \in P\) accepts at most one REQUEST(update\(_i\)(c)) for any c in an execution.

\[\square\]

**NV1** "Let o be a scan\(_i\)() operation that returns C. Then for each \(c \in C\), an update\(_j\)(c) operation is invoked by some process \(p_j\) prior to the completion of o." [3]

**Proof.** We proof integrity by contradiction assuming that scan\(_i\)() operation o returns C and there exist a \(c' \neq \bot\) and \(c' \in C\) for which no update\(_q\)(c') operation is invoked by some process \(p_q\) prior to the completion of o.
The value $c' \in C$ means that at least $3f_P + 1$ processes $p_k \in P$ send $<\text{REPLY COLLECT}, \text{Mem}, p_k>$ to $p_i$ (because there are at most $f_P$ byzantine nodes) and at least $f_P + 1$ processes $p_l$ have $c' = \text{Mem}[q].\text{Read}()$ for any $q \in \text{range}(n_P)$. $p_l$ waits for $<\text{OK RECEIVED}, update_q(c'), p_l>$ from at least $f_P + 1$ processes $p_l \in P$. This ensures that at least one $<\text{OK RECEIVED}, update_q(c'), p_c>$ from a correct node $p_c$ has been received by $p_i$. This implies that at least $3f_P + 1$ processes $p_k \in P$ have sent $<\text{WRITTEN}(p_k, update_q(c'))>$ to all $p_r \in P$ and thus, $p_q$ has accepted $<\text{REQUEST}(update_q(c'))>$ and has sent $<\text{OK, update_q(c')}>$ to all $p_k \in P$.

This implies that an $update_q(c')$ operation must be invoked by some process $p_q$ prior to the completion of $o$. So it contradicts our assumption and proofs integrity.

\hfill $\Box$

NV2 "Let $o$ be an $\text{scan}_i()$ operation that is invoked after the completion of an $update_j(c)$ operation, and that returns $C_i$. Then $C_i \neq \emptyset."$ [3]

Proof. Since $update_j(c)$ completes, either (case 1) $\text{collect}()$ procedure during the $update_j(c)$ operation has returned an empty set and $<\text{WRITTEN}(p_k, update_j(c'))>$ from $3f_P + 1$ processes $p_q \in P$ are received by $p_j$ or (case 2) $\text{collect}()$ returns a non-empty set during the $update_j(c)$ operation.

We start with the first case where $o$ invokes a $\text{scan}_i()$ operation. In the $\text{scan}_i()$ operation $o$, $o$ invokes $\text{collect}()$ twice where both times at least $f_P + 1$ processes $p_k \in P$ have $c = \text{Mem}[j].\text{Read}()$ by property of Mem array. Thus, $\text{collect}()$ returns $C_i$ with $c \in C_i$.

The second case is that $\text{collect}()$ completes returning a non-empty set. Thus, at least $2f_P + 1$ processes $p_k \in P$ must have $\text{Mem}[z].\text{Read}() = c'$ for some $z \in \text{range}(n_P)$ with $c' \neq \perp$ such that at least $f_P + 1$ processes $p_l$ returning $<\text{REPLY COLLECT}, \text{Mem}, p_l>$ to $p_l$ have $\text{Mem}[z].\text{Read}() = c'$. By atomicity of Mem array of each $p_k \in P$ and by property of Mem array, since $o$ is invoked after $update_j(c)$ completes, at least $f_P + 1$ processes $p_k \in P$ must have read $\text{Mem}[j].\text{Read}() = c'$ and $\text{collect}()$ returns $C_i$ with $c' \in C_i$.

Thus, in both cases the first and second collect during the $\text{scan}_i()$ operation $o$ return a non-empty set, which means that $C_i \neq \emptyset$.

\hfill $\Box$

NV3 "Let $o$ be a $\text{scan}_i()$ operation that returns $C_i$ and let $o'$ be a $\text{scan}_j()$ operation that returns $C_j$ and is invoked after the completion of $o$. Then $C_i \subseteq C_j.$" [3]
3. The Weak Snapshot Abstraction

Proof. If \(C_i = \emptyset\), the lemma trivially holds. Otherwise, consider any \(c \in C_i\). \(c \in C_i\) can be achieved by the second \texttt{collect()} of \(o\) by receiving \texttt{<REPLY COLLECT, Mem, \(p_k\)}, from \(3f_P+1\) processes \(p_k \in P\) and at least \(f_P+1\) processes \(p_l \in P\), which have sent \texttt{<REPLY COLLECT, Mem, \(p_l\)}, to \(p_i\), have \texttt{Mem[q].Read()}=c for any \(q \in \text{range}(nP)\). Due to the fact that at least \(f_P+1\) processes \(p_l \in P\), which have sent \texttt{<REPLY COLLECT, Mem, \(p_l\)}, to \(p_j\), have \texttt{Mem[q].Read()}=c for any \(q \in \text{range}(nP)\), \(p_i\) waits for \texttt{<OK RECEIVED, \texttt{update}_q(c)}, \(p_n\) from at least \(f_P+1\) processes \(p_n \in P\).

We now prove that \(c \in C_j\) also holds which can be achieved by receiving \texttt{<REPLY COLLECT, Mem, \(p_k\)} from \(3f_P+1\) processes \(p_k \in P\) and at least \(f_P+1\) processes \(p_l \in P\), which have sent \texttt{<REPLY COLLECT, Mem, \(p_l\)} to \(p_j\), have \texttt{Mem[q].Read()}=c for any \(q \in \text{range}(nP)\).

During the second \texttt{collect()} of \(o\), \(p_i\) waits for \texttt{<OK RECEIVED, \texttt{update}_q(c)}, \(p_n\) from at least \(f_P+1\) processes \(p_n \in P\). This implies that at least one \texttt{<OK RECEIVED, \texttt{update}_q(c)}, \(p_c\) from a correct node \(p_c\) has been received by \(p_i\). Thus, the \texttt{\texttt{update}_q(c)} has terminated. Therefore, \(p_c\) has waited in the \texttt{\texttt{update}_q(c)} operation for \texttt{<WRITTEN}(p_k, \texttt{\texttt{update}_q(c)}) from \(3f_P+1\) processes \(p_k \in P\) and at most \(f_P\) byzantine processes have sent \texttt{<WRITTEN}(p_k, \texttt{\texttt{update}_q(c)}) to all \(p_r \in P\) without having issued \texttt{Mem[q].Write}(c). It follows that at least \(2f_P+1\) processes \(p_m\) have \texttt{Mem[q].Read()}=c.

In the \texttt{\texttt{scan}_j()} operation \(o'\), \(p_j\) receives \texttt{<REPLY COLLECT, Mem, \(p_k\)} from \(3f_P+1\) processes \(p_k \in P\) (because there are at most \(f_P\) byzantine nodes). Due to the fact that \(o'\) is invoked after \(o\) completes, it holds that both times \texttt{collect()} is executed during \(o'\), at least \(f_P+1\) processes \(p_l \in P\), which have sent \texttt{<REPLY COLLECT, Mem, \(p_l\)} to \(p_j\), have \(c=\texttt{Mem[q].Read()}\) for any \(q \in \text{range}(nP)\) and \(c \neq \bot\). The reason for this is that there are at most \(f_P\) correct nodes and at most \(f_P\) byzantine nodes returning \texttt{<REPLY COLLECT, Mem, \(p_k\)} containing a Mem array with \texttt{Mem[q].Read()}\neq c for any \(q \in \text{range}(nP)\). This follows from the fact that in the \texttt{\texttt{update}_q(c)} operation at most \(f_P\) byzantine nodes can send \texttt{<WRITTEN}(p_k, \texttt{\texttt{update}_q(c)}) to all \(p_r \in P\) without having issued \texttt{Mem[q].Write}(c) such that at most \(f_P\) correct nodes have not issued \texttt{Mem[q].Write}(c) and have not sent \texttt{<WRITTEN}(p_k, \texttt{\texttt{update}_q(c)}) to all \(p_r \in P\).

Since \(o'\) is invoked after \(o\) completes both times a set \(C_j\) containing \(c\) is returned from \texttt{collect()}. This implies that \(c \in C_j\) and thus, \(C_i \subseteq C_j\).

\[\square\]

\textbf{NV4} "There exists a \(c\) such that for every \texttt{\texttt{scan}_j()} operation that returns \(C_j \neq \emptyset\), it holds that \(c \in C_j\)." [3]

Proof. Let \(o\) be the first \texttt{\texttt{scan}_k()} operation during which the collect procedure in Algorithm 1 page 9 line 28 returns a non-empty set, and let \(C_i \neq \emptyset\) be this set. Let \(o'\) be any \texttt{\texttt{scan}_j()} operation that returns \(C_j \neq \emptyset\). We next show that
3. The Weak Snapshot Abstraction

\[ C_i \subseteq C_j, \] which means that any \( c \in C_i \) preserves the requirements of the lemma. Since \( C_j \neq \emptyset \), the first invocation of \( \text{collect}() \) during \( o' \) returns a non-empty set. By definition of \( o \), the second \( \text{collect}() \) during \( o' \) starts after the first \( \text{collect}() \) of \( o \) completes.

For every \( c \in C_i \), at least \( 3f_P + 1 \) processes \( p_k \in P \) send \( \langle \text{REPLY COLLECT, Mem, } p_k \rangle \) to \( p_i \) during the first \( \text{collect}() \) of \( o \). For every \( c \in C_i \) and any \( q \in \text{range}(nP) \) it holds that at least \( f_P + 1 \) processes \( p_l \in P \), which have sent \( \langle \text{REPLY COLLECT, Mem, } p_l \rangle \) to \( p_i \), have \( \text{Mem}[q].\text{Read}() = c \). Therefore, \( p_i \) waits for \( \langle \text{OK RECEIVED, update}_q(c), p_n \rangle \) from at least \( f_P + 1 \) processes \( p_n \in P \).

We now proof that \( c \in C_j \) also holds which can be achieved by receiving \( \langle \text{REPLY COLLECT, Mem, } p_k \rangle \) from \( 3f_P + 1 \) processes \( p_k \in P \) and at least \( f_P + 1 \) processes \( p_l \in P \), which have sent \( \langle \text{REPLY COLLECT, Mem, } p_l \rangle \) to \( p_j \), have \( \text{Mem}[q].\text{Read}() = c \) for any \( q \in \text{range}(nP) \).

During the first \( \text{collect}() \) of \( o \), \( p_i \) waits for \( \langle \text{OK RECEIVED, update}_q(c), p_n \rangle \) from at least \( f_P + 1 \) processes \( p_n \in P \). This implies that at least one \( \langle \text{OK RECEIVED, update}_q(c), p_c \rangle \) from a correct node \( p_c \) has been received by \( p_i \). Thus, the \( \text{update}_q(c) \) has terminated. Therefore, \( p_q \) has waited in the \( \text{update}_q(c) \) operation for \( \langle \text{WRITTEN}(p_k, \text{update}_q(c)) \rangle \) to \( \text{Mem}[q].\text{Read}() = c \) for any \( q \in \text{range}(nP) \) and \( c \neq \bot \). The reason for this is that there are at most \( f_P \) Byzantine nodes and at most \( f_P \) correct nodes returning \( \langle \text{REPLY COLLECT, Mem, } p_l \rangle \) containing a Mem array with \( \text{Mem}[q].\text{Read}() \neq c \) for any \( q \in \text{range}(nP) \).

In the \( \text{scan}_j() \) operation \( o' \), \( p_j \) receives \( \langle \text{REPLY COLLECT, Mem, } p_k \rangle \) from \( 3f_P + 1 \) processes \( p_k \in P \) (because there are at most \( f_P \) Byzantine nodes). Due to the fact that \( o' \) is invoked after the first \( \text{collect}() \) of \( o \) completes, it holds that both times \( \text{collect}() \) is executed during \( o' \), at least \( f_P + 1 \) processes \( p_l \in P \), which have sent \( \langle \text{REPLY COLLECT, Mem, } p_l \rangle \) to \( p_j \), have \( c = \text{Mem}[q].\text{Read}() \) for any \( q \in \text{range}(nP) \) and \( c \neq \bot \). The reason for this is that there are at most \( f_P \) Byzantine nodes and at most \( f_P \) correct nodes returning \( \langle \text{REPLY COLLECT, Mem, } p_l \rangle \) containing a Mem array with \( \text{Mem}[q].\text{Read}() \neq c \) for any \( q \in \text{range}(nP) \). This follows from the fact that in the \( \text{update}_q(c) \) operation at most \( f_P \) Byzantine nodes can send \( \langle \text{WRITTEN}(p_k, \text{update}_q(c)) \rangle \) to all \( p_r \in P \) without having issued \( \text{Mem}[q].\text{Write}(c) \) and have not sent \( \langle \text{WRITTEN}(p_k, \text{update}_q(c)) \rangle \) to all \( p_r \in P \).

Since \( o' \) is invoked after \( o \) completes both times a set \( C_j \) containing \( c \) is returned from \( \text{collect}() \). Thus, the second \( \text{collect}() \) of \( o' \) returns \( c \) and this implies that \( c \in C_j \) and \( C_i \subseteq C_j \).

\[ NV5 \] "If some majority \( M \) of processes in \( P \) keep taking steps then every \( \text{scan}_i() \) and \( \text{update}_i(c) \) invoked by every \( p_i \in M \) eventually completes." [3]
Proof. We first show that the collect() procedure eventually completes.

In the collect() procedure, \( p_i \) sends \(<\texttt{COLLECT}>\) to all \( p_k \in P \) and waits for \(<\texttt{REPLY COLLECT}, \text{Mem}, p_k>\) from at least \( 3f_P + 1 \) processes \( p_k \). Each of the \( 3f_P + 1 \) correct processes \( p_k \) will eventually receive the \(<\texttt{COLLECT}>\) from \( p_i \) and therefore \( p_k \) sends \(<\texttt{REPLY COLLECT}, \text{Mem}, p_k>\) to \( p_i \). Thus, process \( p_i \) will eventually receive \(<\texttt{REPLY COLLECT}, \text{Mem}, p_k>\) from at least \( 3f_P + 1 \) processes \( p_k \in P \).

If \( p_i \) had received \(<\texttt{REPLY COLLECT}, \text{Mem}, p_l>\) from \( f_P + 1 \) processes \( p_l \) with \( c = \text{Mem}[q].\text{Read()} \) for any \( q \in \text{range}(n_P) \), then \( p_i \) would wait for \(<\texttt{OK RECEIVED}, \text{update}_q(c), p_n>\) from at least \( f_P + 1 \) processes \( p_n \in P \).

Due to the fact that at least one correct node has \( c = \text{Mem}[q].\text{Read()} \) for some \( q \in \text{range}(n_P) \), it holds that at least one correct node \( p_c \) has issued \( \text{Mem}[q].\text{Write}(c) \) and accepted \( \text{REQUEST}(\text{update}_q(c)) \). According to the asynchronous reliable broadcast algorithm in section 3.1 property 3 if a correct node accepts a message, then the message will be eventually accepted by every correct node. This implies that every correct node \( p_u \) will eventually send a \(<\texttt{WRITTEN}(p_u, \text{update}_q(c))>\) to all \( p_k \in P \). Hence, \( p_q \) will eventually receive \(<\texttt{WRITTEN}(p_k, \text{update}_q(c))>\) from \( 3f_P + 1 \) processes \( p_k \in P \) and sends an \(<\texttt{OK, update}_q(c)>\) to all \( p_n \in P \). If \( p_n \) had received \(<\texttt{OK, update}_q(c)>\), then \( p_n \) would wait for \(<\texttt{WRITTEN}(p_k, \text{update}_q(c))>\) from at least \( 3f_P + 1 \) processes \( p_k \in P \). This condition can be satisfied because as stated above every correct node \( p_u \) will send \(<\texttt{WRITTEN}(p_u, \text{update}_q(c))>\) to all \( p_r \in P \). Afterwards, \( p_n \) will send \(<\texttt{OK RECEIVED, update}_q(c), p_n>\) to all \( p_k \in P \). \( p_i \) will eventually receive \(<\texttt{OK RECEIVED, update}_q(c), p_n>\) from at least \( f_P + 1 \) processes \( p_n \in P \) because each of the \( 3f_P + 1 \) correct node \( p_u \in P \) will eventually send \(<\texttt{OK RECEIVED, update}_q(c), p_n>\) to all \( p_k \in P \). Thus, the liveness condition is satisfied.

If \( p_i \) had not received \(<\texttt{REPLY COLLECT}, \text{Mem}, p_l>\) from \( f_P + 1 \) processes \( p_l \) with \( c = \text{Mem}[q].\text{Read()} \) for some \( q \in \text{range}(n_P) \), the liveness condition could be satisfied directly because \( p_l \) does not issue any wait condition.

Thus, the collect() procedure eventually completes.

We now show that the \( \text{update}_i(c) \) operation eventually completes knowing that collect() procedure eventually completes.

On condition that the collect() procedure returns a non-empty set, the \( \text{update}_i(c) \) operation directly sends \(<\texttt{OK, update}_i(c)>\) to all \( p_k \in P \) indicating that it has successfully completed.

After the collect() procedure returns an empty set, process \( p_i \) sends \(<\texttt{REQUEST}(\text{update}_i(c))>\) to all \( p_k \in P \). Process \( p_i \) sends an \(<\texttt{ECHO}(p_t, \text{request}(\text{update}_i(c)))>\) to all \( p_k \in P \) because \( p_t \) sends the \(<\texttt{REQUEST}(\text{update}_i(c))>\) and therefore \( p_t \) has already received \(<\texttt{REQUEST}(\text{update}_i(c))>\). If \( p_i \) had not sent the \(<\texttt{ECHO}(p_k, \text{request}(\text{update}_i(c)))>\) to all \( p_k \in P \) and \( p_i \) is a correct node, no process \( p_r \in P \) could receive \(<\texttt{ECHO}(p_k, \text{request}(\text{update}_i(c)))>\) from
3. The Weak Snapshot Abstraction

$3f_P + 1$ processes $p_k \in P$. The reason for this is that we have $f_P$ byzantine nodes and only $3f_P$ correct nodes $p_k$ would send an $<\text{ECHO}(p_k, \text{request}(\text{update}_i(c)))>$. Therefore, $p_i$ would get stuck waiting for $<\text{ECHO}(p_k, \text{request}(\text{update}_i(c)))>$ from at least $3f_P + 1$ processes $p_k$. Thus, $\text{update}_i(c)$ would not be able to eventually complete.

Each process $p_q \in P$ will eventually receive $<\text{REQUEST}(\text{update}_j(c))>$ from $p_j$ for the first time or $<\text{ECHO}(p_k, \text{request}(\text{update}_j(c)))>$ from $f_P + 1$ processes $p_k \in P$ for the first time because each of the $3f_P + 1$ correct node $p_l \in P$ will eventually receive $<\text{REQUEST}(\text{update}_j(c))>$ from $p_j$. Then, $p_l$ sends $<\text{ECHO}(p_l, \text{request}(\text{update}_j(c)))>$ to all $p_k \in P$.

$p_i$ will eventually receive $<\text{ECHO}(p_k, \text{request}(\text{update}_i(c)))>$ from $3f_P + 1$ processes $p_k$ because of $3f_P + 1$ correct nodes. Then $p_i$ sends $<\text{WRITTEN}(p_k, \text{update}_i(c))>$ to all $p_k \in P$.

$p_i$ waits for $<\text{WRITTEN}(p_k, \text{update}_i(c))>$ from $3f_P + 1$ processes $p_k$ because there are at most $f_P$ byzantine nodes and each of the $3f_P + 1$ correct node $p_q \in P$ will eventually send $<\text{WRITTEN}(p_q, \text{update}_j(c), c)>$ to all $p_r \in P$. Therefore, $p_i$ will eventually receive $<\text{WRITTEN}(p_k, \text{update}_j(c), c)>$ from $3f_P + 1$ processes $p_k \in P$.

Finally, $p_i$ sends $<\text{OK}, \text{update}_i(c)>$ to all $p_k \in P$ to indicate that the $\text{update}_i(c)$ operation eventually completes.

We now show that the $\text{scan}_i()$ operation eventually completes.

In the $\text{scan}_i()$ operation, we issue the $\text{collect()}$ procedure. Since we have proven that the $\text{collect()}$ procedure eventually completes, the $\text{scan}_i()$ operation also eventually completes.
4.1 Conclusion

In this thesis, as a first step we specified the system model and specification in which our weak snapshot abstraction operates. This system model is a distributed system without a consensus guarantee. Our weak snapshot abstraction is implemented based on a collection of processes that interact utilizing asynchronous message passing. Furthermore, a set of processes $P$, where at most $f_P$ out of $n_P=4f_P+1$ processes in $P$ are byzantine, can access a weak snapshot object $S$. Our proposed weak snapshot abstraction is different from previous works as we improved the robustness and the reliability of the weak snapshot abstraction by allowing byzantine processes in the system.

In Chapter 3, we designed a weak snapshot algorithm consisting of two operations, namely $\text{scan}_i()$ operation and $\text{update}_i(c)$ operation. For the set $C_i$ returned by a $\text{scan}_i()$ operation, we require that when a scan sees any updates, then it sees the "first" update. Additionally, we define the $\text{scan}_i()$ operation $o$ to be the first to complete its first collect(). This implies that any $\text{scan}_k()$ operation $o'$ starts its second collect() only after $o$ completes its first collect(). To implement the $\text{update}_i(c)$ operation, we utilized a slightly modified version of the asynchronous reliable broadcast algorithm. Our proposed weak snapshot algorithm fulfills the semantics specified in section 3.2.1 and the property of Mem array which demands that each register Mem[$i$] contains at most one value which is not $\perp$. We proved this in section 3.2.6.

4.2 Future Work

The developed weak snapshot abstraction can be used as an underlying building block to implement the reconfiguration operation in asynchronous dynamic byzantine message passing systems where consensus is not guaranteed. In the following paragraphs, we outline some possible future applications of our weak snapshot abstraction.
The system requirements of our weak snapshot abstraction are based on the work [3], which solves the atomic R/W storage problem in a dynamic setting without consensus or stronger primitives. Therefore, our weak snapshot abstraction can be utilized as a basis to design a reconfiguration operation that solves the atomic R/W storage problem in a dynamic setting without consensus or stronger primitives with at most $f_P$ byzantine processes out of a set of processes $P$ of size $n_P=4f_P+1$. The set of processes $P$ can access a weak snapshot object $S$.

Besides, as our proposed weak snapshot abstraction operates in asynchronous dynamic byzantine message passing systems where consensus is not guaranteed and provides an $update_i(c)$ and a $scan_i()$ operation, it can be used as a basis to develop and establish a reconfiguration operation in an asynchronous dynamic byzantine reliable broadcast algorithm.
Bibliography

