Partitionable Virtual Synchrony

Using

Extended Virtual Synchrony

John Lane Schultz

A thesis submitted to The Johns Hopkins University in conformity with the requirements for the degree of Master of Science, Engineering.

Baltimore, Maryland

January 2001

Abstract

View-oriented group communication systems (GCSs) are powerful tools for building distributed applications. Over the past fifteen years, group communication researchers developed a multitude of group communication semantics and implementations. Today, researchers commonly design their group communication algorithms on top of simple existing services such as a network membership service or a reliable FIFO multicast framework. A natural extension of this idea is to implement one set of group communication semantics using another. This approach is not usually utilized due to the expensive overhead of running one set of group communication algorithms on top of another.

This thesis argues that the Extended Virtual Synchrony (EVS) model of group communication, implemented using a client-daemon architecture, is of such high performance that the overhead of constructing another group communication model on top of it is acceptable. It demonstrates that the strong safety properties provided by the EVS model can be leveraged to create very simple algorithms that implement more powerful group communication models.

This thesis presents several EVS algorithms for implementing a partitionable Virtual Synchrony (VS) model of group communication. It first explicitly defines the VS and EVS models through the presentation of their safety and liveness properties. Then, one simple algorithm is formally proved to implement the VS model by utilizing the safety and liven ess properties of the underlying EVS system. Finally, the paper discusses several other simple variants and algorithms that were developed during the course of this work.

Acknowledgments

First and foremost, I thank my mother, Gloria Lane Single, for nurturing and supporting me for the last twenty-three years of my life. My mother truly is the bedrock on which my life has been founded, and all of my achievements stem from her. My stepfather, George Thomas Single, has been in my life since before I can remember. Throughout my life “Mr. Tom” has shown me, by example, how one should live up to their responsibilities; I hope that I can become half the man that he is.

I am deeply and permanently indebted to my advisor, Dr. Yair Amir, for everything he has done for me during my time at Hopkins. Without his persuasion and support I would not have pursued a Master’s degree when I did. I believe I learned more in the pursuit of this research than I did in my entire course work at Hopkins. I also thank Yair for his help in creating my first commercial venture, D-Fusion, Inc.; without his assistance, D-Fusion would never have gotten off the ground. Finally, I thank Yair for reminding me about what is really important in life and making sure I keep my eyes on the bigger picture.

I am grateful to my father, Carl Matthew Schultz, Jr., and stepmother, Claudette Schultz, for all of their emotional and financial support. Without them, I would never have been able to attend Hopkins. I believe that my study at Hopkins will drastically affect the rest of my life for the better and I am deeply indebted to them for that.

I thank Jonathan Stanton for his gracious ability to get involved with almost any problem that was bothering me, regardless of topic. Jonathan makes up a big portion of the lab and will be sorely missed when he moves on. I thank Cristina Nita-Rotaru for putting up with me while we worked together and for writing such masterful publications. I thank each and every other person in The Center for Networking and Distributed Systems for the discussions, debates and brainstorming sessions that we had with one another. I can honestly say that they are the best group of people with which I have ever worked.

I thank Katy Hsieh for giving me a taste of true love and becoming one of my closest friends. I thank my friends Bobby Day, Phil Vasic and especially Eric Hall for remaining true friends and tolerating my inexcusably long absences while I was at Hopkins.

Finally, I thank Jacob Green for being a close and true friend ever since I first met him. I thank him for occasionally dragging me out of the house to have some fun. I thank him for his tireless perseverance and cajoling in the pursuit of our first joint venture, D-Fusion. I look forward to a lifetime of friendship and business partnership with Jacob.

John Lane Schultz
January 2001

My work in The Center for Networking and Distributed Systems was supported by the National Security Agency under the LUCITE program and also by The Department of Computer Science, The Johns Hopkins University.

Table of Contents

1 Introduction

View-oriented group communication systems (GCSs) are powerful tools that can greatly simplify the development of distributed systems and services. GCSs provide two interrelated services to their clients: a membership service and a multicast service. The multicast service allows client processes to intercommunicate by multicasting datagram messages, while the membership service tracks and reports the set of currently connected clients with which a client can communicate. The exact semantic guarantees of the membership and multicast services are specified by the particular GCS's safety and liveness properties.

Group communication is an active area of research that has been under development for more than fifteen years [Bir86]. During that time, researchers proposed many different systems that offered tradeoffs between performance, fault-tolerance and semantic guarantees (see [VKCD99] for comprehensive references). With the multitude of available semantics and implementations, it becomes interesting to look at how these different systems are realized. Almost all of these systems can be built "from scratch" using only an unreliable packet service such as UDP or even IP [MPS91, ADKM92, BvR94, HvR96, AS98]. However, it usually does not make sense for a GCS researcher to "reinvent the wheel" in this manner every time. Instead, GCS researchers will often design their algorithms on top of simple services such as an existing membership service or a reliable FIFO multicast framework (e.g. [KK00]).

A natural extension of this idea is to implement one set of GCS semantics using another. This would allow a designer to leverage all of the strong services provided by the underlying GCS's semantics in order to develop simpler algorithms. Researchers, however, do not normally take this approach due to the excessive overhead of implementing one GCS algorithm on top of another. In most cases, this supposition is correct. The Extended Virtual Synchrony (EVS) model [MAMSA94, Ami95], when implemented using a client-daemon model [AS98], however, provides uniquely high performance services to its clients. This high-performance GCS allows a stronger set of GCS semantics to be built on top of it without excessive overhead.

The main objective of this thesis is to explore the simple and effective implementation of a stronger set of GCS semantics built on top of EVS semantics. This exploration is done using the Virtual Synchrony (VS) model [GS95], one of the best-understood and most-used group communication models. This thesis demonstrates that a partitionable Virtual Synchrony model can be implemented effectively on top of the Extended Virtual Synchrony model with relatively simple and efficient algorithms.

The major contributions of this work are: it presents (1) precise specifications of the VS and EVS models using I/O automata and mathematical notation, (2) an EVS algorithm that implements the VS model of group communication, (3) rigorous proofs of that algorithm's correctness and (4) it allows EVS systems to support the VS model with a simple client module.

1.1 Outline

This thesis is divided into three main parts: the first part presents the VS and EVS models, the second presents an algorithm that implements VS on top of EVS and proves the algorithm's correctness and the last part discusses several algorithm optimizations and model variants along with their respective tradeoffs.

Section 2 gives a brief overview of related work.

Section 3 states and discusses the safety properties that specify the VS and EVS models for the purposes of this thesis. [MAMSA94] specified the canonical definition of EVS. The VS model, on the other hand, has no canonical definition and the usage of this name is somewhat confusing throughout the literature. By defining the exact VS model used in this thesis there should be no further confusion generated by this work.

Section 4 lays out the problem of implementing VS on top of EVS and briefly discusses some membership liveness properties that most GCSs maintain. This section explains the general approach and algorithm this work used for implementing VS on top of EVS.

Section 5 presents an algorithm in pseudo-code that implements the presented VS model on top of the presented EVS model.

Section 6 formally proves that the presented algorithm correctly implements the VS model by leveraging the safety and liveness properties of the underlying EVS model.

Section 7 discusses several algorithmic and model variants and discusses their respective tradeoffs.

Section 8 presents some real-world performance statistics from one of the algorithm variants that were developed by this work.

Finally, section 9 concludes this thesis and summarizes the contributions of this work.

2 Related Work

Reliable group communication is an active research area, rich with specifications, implementations and applications. In the past, most works concentrated on the performance and capabilities of systems, often with a particular application in mind. Several different group communication systems were built, such as ISIS [BvR94], Horus [RKM96], Transis [ADKM92], Totem [AMMS$^+$95], RMP [WMK94] and Spread [AS98]. All of these systems are based on the ideas of virtual synchrony [Bir86] and generally provide different “flavors” of the two most popular semantic models: Virtual Synchrony [BvR94] and Extended Virtual Synchrony [MAMSA94].

Recently, precise specifications of system properties with accompanying proofs of correctness have become more important. Researchers aided this movement by developing several formal specification systems well suited to modeling distributed systems, such as the I/O automaton paradigm [LT89, Lyn96, GL98] and Vitenberg’s multi-sorted algebra [Vit98]. These systems allow for precise and easy to understand property specifications of distributed systems and have been used recently for specifying and reasoning about GCSs [DPFLS98, KK00]. These specification systems allow reasoning about composition and arbitrary combinations of properties in an unambiguous manner. In addition, these systems can lead to modular specifications, which can easily translate to modular or layered system designs. Automatic theorem proving tools have also been developed to work with these types of specification systems, such as the Larch Prover [GG91, GHG$^+$93] and have been used to prove the correctness of several algorithms [SAGG$^+$93, PPG$^+$96, LSGL95].

In [VKCD99], the I/O automaton model was used to specify a host of logic formulae specifying most common group communication safety and liveness properties. That seminal work laid the groundwork for formal specifications of group communication systems for the future. This paper strongly adopts and endorses the use of their specification style. It allows for unambiguous and clear statements of system properties that lend themselves both to manual and automatic proofs.

Other related work is sited throughout this thesis in the particular sections where that related work is most pertinent.

3 The VS and EVS Group Communication Models

This section specifies the Virtual Synchrony (VS) and Extended Virtual Synchrony (EVS) models in four subsections. The first subsection discusses the formalism used for specifying the models and the following three subsections actually present the two sets of safety properties. The second subsection presents the commonalities that the two models share, the third subsection presents the additional properties that EVS provides and the last subsection presents the additional properties that VS provides.

3.1 Presentation Formalism

This thesis presents a GCS model as a set of safety and liveness properties that defines the system's membership and multicast services. This thesis adopts and relies heavily on the specification style of [VKCD99], where properties are formalized as trace properties of an I/O automaton [LT89] in logical axioms using set-theoretic notation. This thesis precisely specifies the safety properties of each model while it informally discusses the liveness properties of group communication systems. This thesis concentrates on formally proving that the discussed algorithms maintain the models' safety properties while maintaining their liveness properties without formal proofs. For explicit specifications and more in-depth discussions of GCS liveness properties, please consult [VKCD99].

In this thesis, GCS processes are modeled as untimed I/O automata [LT89, Lyn96]. The safety and liveness properties presented herein are with respect to the external behavior of the GCS processes, as reflected in their external signature and in their fair traces. The external signature of an automaton consists of the possible atomic input and output actions with which it can interact with its environment. A trace of an I/O automaton is the sequence of external actions that occur at that automaton in an execution. An automaton is said to implement a GCS model if it has the same external signature as that model and for all of that automaton's fair traces the safety and liveness properties of that model hold. For formal definitions and a more in-depth discussion of I/O automatons, please consult [Lyn96], Chapter 8.

The GCSs that this work considers function on top of a communication network that provides asynchronous, unreliable message delivery. The group communication models allow for the following external events: messages may be lost, processes may crash and recover, the network can partition into disjoint network components and previously disjoint network components may merge. The models considered herein are partitionable, meaning that a client process can make progress in any network component. This thesis assumes that no Byzantine failures occur, meaning that no client or component of the GCS processes acts in a non-specified manner.

For simplicity's sake, this thesis makes the following assumptions in its presentations: (1) it is assumed there exists one GCS process for each client process and they are, in fact, one in the same, therefore, if any part of the GCS process or client fails, this results in a GCS process crash; (2) all properties and algorithms presented in this thesis are with respect to a single client group, whereas most GCS systems provide multiple groups to and from which clients can send and receive messages; (3) that group is a closed group, meaning that only members of that group can send and receive messages to and from it; (4) the mechanisms of joining and leaving that group are considered external operations – upon recovery, a process immediately tries to join the group and only leaves that group as a result of crashing. None of these assumptions have important side effects and they can be removed. These assumptions are only used to simplify the presentation of the models, algorithms and proofs.

3.2 Shared GCS Model

This section presents the portions of the VS and EVS models that they share in common.

3.2.1 Automaton External Signature

The specifications of the GCS models use the following basic sets:

B – the Boolean set
N – the set of natural numbers
P – the set of processes
M – the set of sent client messages
VID – the set of delivered view ids with a strict partial order by the $<$ operator
MT := { R, F, C, A, S } the set of messages types
$\varnothing$ – the null or empty set

Throughout the rest of this thesis, variables named a, b, c, d, i, j, k and l are members of N, variables named p and q are members of P, variables named D, S and T are members of 2^P, variables named m are members of M and variables named id are members of VID. Any variable name that has a prime(s) or star(s) appended to it is from the same set as the base variable.

The basic functionality or signature of a view-oriented GCS is that it allows a client to send messages to other clients, receive messages from other clients, and get information about the other clients with which it is communicating.

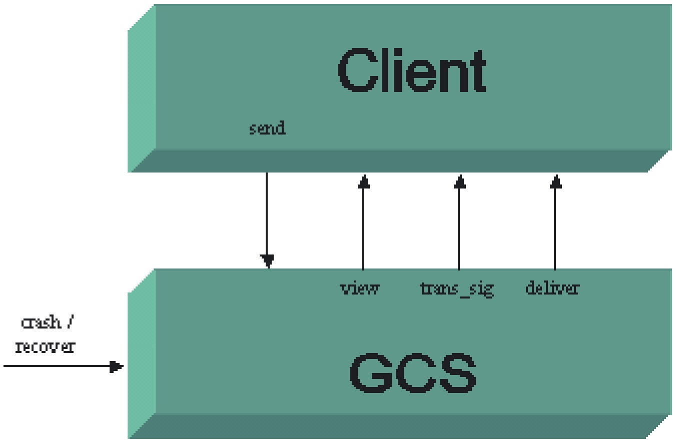


Figure 1: External signature of the Shared GCS Model.

Each action in the GCS external signature is parameterized by a unique p rocess $p \in P$ at which that action occurs. Each GCS process interacts with its client and environment as depicted in Figure 1. The external signature of the GCS consists of the following actions:

Interaction with the Client

- input **send**(p, m), $p \in P$, $m \in M$

  Note that each sent message is associated with one sender and one message type – this information is assumed to be encoded with the message. I refer to the message sender as m.sender ($\in P$) and the message type as m.type ($\in MT$).

- output **deliver**(p, m), $p \in P$, $m \in M$
- output **view**(p, id, D, T), $p \in P$, $id \in VID$, $D \in 2^P$, $T \in 2^P$

  D represents the membership set of the view and T represents the transitional set of the view for this process.

- output **trans_sig**(p), $p \in P$

Interaction with the Environment

- input **crash**(p), $p \in P$
- input **recover**(p), $p \in P$

3.1 **Definition (Event)** *An event is an occurrence of an action from an automaton's external signature.*

3.2 **Definition (Trace)** *A trace is a sequence of events.*

3.2.2 Mathematical Model

This section presents the mathematical model for stating trace properties of a GCS automaton with the external signature above described. The properties are stated in logical axioms using set-theoretic notation and use the following sets:

B, N, P, M, VID, MT, $\varnothing$ – the basic sets above described

Actions The set of actions is:

$$\{ \textbf{send}(p, m) \mid p \in P, m \in M \} \cup \{ \textbf{deliver}(p, m) \mid p \in P, m \in M \} \cup$$
$$\{ \textbf{view}(p, id, D, T) \mid p \in P, id \in VID, D \in 2^P, \ T \in 2^P \} \cup$$
$$\{ \textbf{trans\_sig}(p) \mid p \in P \} \cup \{ \textbf{crash}(p) \mid p \in P \} \cup \{ \textbf{recover}(p) \mid p \in P \}$$

Traces – sequences of actions

Events – actions that are members of traces

Since all of the following axioms classify automaton traces, they all take a trace as a parameter. For clarity of presentation, the trace parameter is considered implicit and is omitted – all axioms are with respect to a fixed trace (*Trace* = $t_1, t_2, \ldots$). In the following axioms, universal quantifiers are omitted – when a variable is unbound it is universally quantified for the scope of the entire formula.

3.2.3 Definitions

Since each event occurs atomically at a single process, the function $pid : Events \rightarrow P$, which returns the process at which each event occurs, is defined.

3.3 **Definition (pid)** *The pid of an event t_a is the process at which that event occurred. Formally:*

$$\begin{aligned} pid(t_a) := p \text{ if } \ & t_a = \textbf{trans\_sig}(p) \vee t_a = \textbf{crash}(p) \vee t_a = \textbf{recover}(p) \vee \\ & (\exists m : t_a = \textbf{deliver}(p, m) \vee t_a = \textbf{send}(p, m)) \vee (\exists id \exists D \exists T : t_a = \textbf{view}(p, id, D, T)) \end{aligned}$$

In a view-oriented GCS, events occur at processes within the context of views. The function $vid : Events \times P \rightarrow VID \cup \{\perp\}$ returns the view in the context of which an event occurred at a specific process. Note that for a **view** event, it is not the new view introduced, but rather the process' previous view. Up until the first **view** event at a process and immediately after a **crash** event, a process is not considered to be in any view (modeled by $\perp$).

3.4 **Definition (vid)** *The vid of an event t_c at a process p is the view identifier delivered in a* **view** *event t_a at p which precedes t_c such that there are no* **view** *or* **recover** *events between t_a and t_c at p. If there is no such* **view** *event then the vid is the null view identifier, $\perp$. Formally:*

$$\begin{aligned} vid(t_c, p) := \ & id \text{ if } \exists a \nexists b \exists D \exists T : a < b < c \wedge t_a = \textbf{view}(p, id, D, T) \wedge \\ & \qquad (t_b = \textbf{recover}(p) \vee \exists id' \exists D' \exists T' : t_b = \textbf{view}(p, id', D', T')) \\ & \perp \text{ otherwise} \end{aligned}$$

Event t_b is the first event at a process p:

$$\text{first\_event}(t_b, p) \equiv \nexists a : a < b \wedge \text{pid}(t_a) = \text{pid}(t_b) = p$$

Event t_a is the previous event before t_c at process p:

$$\text{prev\_event}(t_a, t_c, p) \equiv a < c \wedge \text{pid}(t_a) = \text{pid}(t_c) = p \wedge \nexists b : \text{pid}(t_b) = p \wedge a < b < c$$

Event t_c is the next event after t_a at process p:

$$\text{next\_event}(t_c, t_a, p) \equiv a < c \wedge \text{pid}(t_a) = \text{pid}(t_c) = p \wedge \nexists b : \text{pid}(t_b) = p \wedge a < b < c$$

Table 1: Shorthand predicates.

The Causal order [Lam78] defines a strict partial order on events in a trace.

3.5 **Definition ($\rightarrow$)** *The $\rightarrow$ relation defines the causally precedes strict partial order on events. Formally:*

$$t_a \rightarrow t_i \equiv (\text{pid}(t_a) = \text{pid}(t_i) \wedge a < i) \vee (t_a = \textbf{send}(p, m) \wedge t_i = \textbf{deliver}(q, m)) \vee$$
$$(\exists b : t_a \rightarrow t_b \wedge t_b \rightarrow t_i)$$

The *ord : M $\rightarrow$ N* function is used by the GCS to determine the delivery order of Agreed messages[1]. This function is not necessarily available to client processes.

3.6 **Definition (ord)** *The ord function is a one-to-one mapping from M to the set of natural numbers that is consistent with the causally precedes strict partial order of* **send** *events. Formally:*

$$\text{ord} : M \rightarrow N \wedge (\text{ord}(m) = \text{ord}(m') \Leftrightarrow m = m') \wedge$$
$$(t_a = \textbf{send}(p, m) \wedge t_i = \textbf{send}(q, m') \wedge t_a \rightarrow t_i \Rightarrow \text{ord}(m) < \text{ord}(m'))$$

Several shorthand predicates are also defined in Table 1.

3.2.4 Assumptions about the Environment

In the following models, no events occur at a process between crash and recovery.

3.1 **Assumption (Execution Integrity)** *The first event that occurs at a process is a* **recover** event. *If a* **recover** *event occurs at a process, then it is either the first event at that process or the previous event was a* **crash** event. *The next event that occurs at a process after a* **crash** *event is a* **recover** *event. Formally:*

$$(\text{first\_event}(t_b, p) \Rightarrow t_b = \textbf{recover}(p)) \wedge$$
$$(t_b = \textbf{recover}(p) \Rightarrow (\text{prev\_event}(t_a, t_b, p) \wedge t_a = \textbf{crash}(p)) \vee \text{first\_event}(t_b, p)) \wedge$$
$$(t_a = \textbf{crash}(p) \wedge \text{next\_event}(t_b, t_a, p) \Rightarrow t_b = \textbf{recover}(p))$$

[1] Agreed messages are delivered in a strong total order [WS95, VKCD99].

In order to distinguish between messages sent in different **send** events, each message sent by a client is tagged with a unique message identifier. This assumption is not essential and is, again, used to simplify the presentation of the models.

3.2 **Assumption (Message Uniqueness)** *There are no two different* **send** *events with the same content. Formally:*

$t_a = \textbf{send}(p, m) \wedge t_i = \textbf{send}(q, m) \Rightarrow a = i$

3.2.5 Shared GCS Membership Service Safety Properties

This section presents the safety properties of the membership service that both the VS and EVS models maintain.

3.1 **Property (Initial View Event)** *Every* **send, deliver,** *and* **trans_sig** *event at a process occurs within some view. Formally:*

$t_a = \textbf{send}(p, m) \vee t_a = \textbf{deliver}(p, m) \vee t_a = \textbf{trans\_sig}(p) \Rightarrow \text{vid}(t_a, p) \neq \perp$

3.2 **Property (Self Inclusion)** *If a process p installs a view, then p is a member of the membership set. Formally:*

$t_a = \textbf{view}(p, id, D, T) \Rightarrow p \in D$

3.3 **Property (Membership Agreement)** *If a process p installs a view with identifier id and a process q installs a view with the same identifier, then the membership sets of the views are identical. Formally:*

$t_a = \textbf{view}(p, id, D, T) \wedge t_i = \textbf{view}(q, id, D', T') \Rightarrow D = D'$

3.4 **Property (Local Monotonicity)** *If a process p installs a view with identifier id' after installing a view with identifier id, then id' is greater than id. Formally:*

$t_a = \textbf{view}(p, id, D, T) \wedge t_b = \textbf{view}(p, id', D', T') \wedge i < j \Rightarrow id < id'$

3.2.6 Shared GCS Multicast Service Safety Properties

This section presents the safety properties of the multicast service that both the VS and EVS models maintain.

3.5 **Property (No Duplication)** *A process never delivers a message more than once. Formally:*

$t_a = \textbf{deliver}(p, m) \wedge t_b = \textbf{deliver}(p, m) \Rightarrow a = b$

3.6 **Property (Delivery Integrity)** *A* **deliver** *event in a view is the result of a preceding* **send** *event by a member of that view*[2]*. Formally:*

$t_a = \textbf{view}(p, id, D, T) \wedge t_b = \textbf{deliver}(p, m) \wedge \text{vid}(t_b, p) = id \Rightarrow \exists i \exists q : i < a \wedge t_i = \textbf{send}(q, m) \wedge q \in D$

[2] Note that the requirement that a sender be a member of the view in which its message is delivered is not required for open group GCSs.

Property (Self-Delivery) is actually a liveness property of the multicast service. It is formally presented here because it is utilized in the section of proofs.

3.7 **Property (Self-Delivery)** *If a process p sends a message m, then p delivers m unless it crashes. Formally:*

$t_a = \textbf{send}(p, m) \wedge \nexists b : a < b \wedge t_b = \textbf{crash}(p) \Rightarrow \exists c : t_c = \textbf{deliver}(p, m)$

3.8 **Property (Same View Delivery)** *If processes p and q both deliver a message m, then they both deliver m in the same view. Formally:*

$t_a = \textbf{deliver}(p, m) \wedge \text{vid}(t_a, p) = \text{id} \wedge t_i = \textbf{deliver}(q, m) \wedge \text{vid}(t_i, q) = \text{id'} \Rightarrow \text{id} = \text{id'}$

The following properties are not usually explicitly stated in specifications of the EVS and VS models. They are specified here because, unlike most specifications, this EVS model does not assume any form of Sending View Delivery (defined below) and, therefore, requires more explicit properties to maintain the usual message reliability (no-holes) safety properties. These properties are sanity constraints on the views in which a process' messages can be delivered. All of the GCSs studied by this work maintain these properties.

3.9 **Property (Sane View Delivery)**

1. *A message is not delivered in a view earlier than the one in which it was sent. Formally:*

   $t_a = \textbf{send}(p, m) \wedge \text{vid}(t_a, p) = \text{id} \wedge t_i = \textbf{deliver}(q, m) \wedge \text{vid}(t_i, q) = \text{id'} \Rightarrow \text{id} \leq \text{id'}$

2. *If a process p sends a message m, crashes and later recovers in a view id and a process q delivers m, then m is delivered in a view before id. Formally:*

   $t_a = \textbf{send}(p, m) \wedge t_c = \textbf{view}(p, \text{id}, D, T) \wedge \text{vid}(t_c, p) = \perp \wedge a < c \wedge t_i = \textbf{deliver}(q, m) \Rightarrow$
   $\text{vid}(t_i, q) < \text{id}$

3. *If two messages m and m' are sent, respectively, by processes p and p' such that the send of m' is causally preceded by the send of m and a process q' delivers both messages, then q' does not deliver m in a later view than m'. Formally:*

   $t_a = \textbf{send}(p, m) \wedge t_d = \textbf{send}(p', m') \wedge t_a \rightarrow t_d \wedge t_j = \textbf{deliver}(q', m) \wedge t_k = \textbf{deliver}(q', m') \Rightarrow$
   $\text{vid}(t_j, q') \leq \text{vid}(t_k, q')$

3.10 **Property (Virtual Synchrony)** *If processes p and q are virtually synchronous in a view (defined below), then any message delivered by p in that view is also delivered by q. Formally:*

$\text{vsynchronous\_in}(p, q, \text{id}) \wedge t_a = \textbf{deliver}(p, m) \wedge \text{vid}(t_a, p) = \text{id} \Rightarrow \exists i : t_i = \textbf{deliver}(q, m)$

This thesis presents the usual message ordering and reliability safety properties differently than most GCS papers. In this thesis, the different message ordering and reliability safety properties are explicitly presented as a hierarchy of different message types. In this hierarchy, each higher level of service maintains all of the safety properties of the lower levels. For example, a FIFO message maintains the FIFO and Reliable message properties, while a Causal message maintains the Causal, FIFO and Reliable message properties. This hierarchy implicitly makes the presented GCSs use *weak incorporated* [WS95] delivery semantics when delivering two different types of messages. (Recall that a sent message m is of one specific message type , indicated by m.type $\in$ MT)

3.11 **Property (Reliable Messages)** *All messages are reliable. The Self-Delivery, Same View Delivery and Virtual Synchrony properties implicitly define the safety properties of Reliable messages. Formally:*

$\text{reliable}(m) \equiv m.\text{type} \in \{ R, F, C, A, S \}$

3.12 **Property (FIFO Messages)**

1. *FIFO messages are reliable messages. Formally:*

   $\text{fifo}(m) \equiv m.\text{type} \in \{ F, C, A, S \}$

2. *If a process sends a FIFO message after sending a previous message, then these messages are delivered in the order in which they were sent at every process that delivers both. Formally:*

   $t_a = \mathbf{send}(p, m) \wedge t_b = \mathbf{send}(p, m') \wedge a < b \wedge \text{fifo}(m') \wedge$
   $t_i = \mathbf{deliver}(q, m) \wedge t_j = \mathbf{deliver}(q, m') \Rightarrow i < j$

3. *If a process p sends a FIFO message m' after sending a previous message m and a process q' delivers m', then if any process delivers m, then q' either delivers m or installs a view without p in its transitional set between the delivery views of m and m', or if no process delivers m, then p crashed between sending m and m' and q' installs a view without p in its transitional set between the recovery view of p and the delivery view of m'.*

   $t_a = \mathbf{send}(p, m) \wedge t_c = \mathbf{send}(p, m') \wedge a < c \wedge \text{fifo}(m') \wedge t_l = \mathbf{deliver}(q', m') \Rightarrow$
   $(\exists i \exists q : t_i = \mathbf{deliver}(q, m) \Rightarrow (\exists j : t_j = \mathbf{deliver}(q', m)) \vee$
   $(\exists k \exists id' \exists D' \exists T' : t_k = \mathbf{view}(q', id', D', T') \wedge p \notin T' \wedge \text{vid}(t_i, q) < id \leq \text{vid}(t_l, q'))) \wedge$
   $(\nexists i \exists q : t_i = \mathbf{deliver}(q, m) \Rightarrow \exists b \exists id \exists D \exists T : a < b < c \wedge t_b = \mathbf{view}(p, id, D, T) \wedge \text{vid}(t_b, p) = \perp \wedge$
   $\exists k \exists id' \exists D' \exists T' : t_k = \mathbf{view}(q', id', D', T') \wedge p \notin T' \wedge id \leq id' \leq \text{vid}(t_l, q'))$

3.13 **Property (Causal Messages)**

1. *Causal messages are FIFO messages. Formally:*

   $\text{causal}(m) \equiv m.\text{type} \in \{ C, A, S \}$

2. *If a process sends a causal message m' such that the send of another message m, causally precedes the send of m', then any process that delivers both messages, delivers m before m'. Formally:*

   $t_a = \mathbf{send}(p, m) \wedge t_d = \mathbf{send}(p', m') \wedge t_a \rightarrow t_d \wedge \text{causal}(m') \wedge$
   $t_i = \mathbf{deliver}(q, m) \wedge t_j = \mathbf{deliver}(q, m') \Rightarrow i < j$

3. *If a process p’ sends a causal message m’ such that the send of another message m causally precedes m’, then if any process delivers m, then q’ either delivers m or installs a view without p’ in its transitional set between the delivery views of m and m’, or if no process delivers m, then p crashed between sending m and m’ and q’ installs a view without p in its transitional set between the recovery view of p and the delivery view of m’.*

$$
\begin{aligned}
&t_a = \mathbf{send}(p, m) \wedge t_c = \mathbf{send}(p', m') \wedge t_a \rightarrow t_c \wedge \text{causal}(m') \wedge t_l = \mathbf{deliver}(q', m') \Rightarrow \\
&(\exists i \exists q : t_i = \mathbf{deliver}(q, m) \Rightarrow (\exists j : t_j = \mathbf{deliver}(q', m)) \vee \\
&\ (\exists k \exists id' \exists D' \exists T' : t_k = \mathbf{view}(q', id', D', T') \wedge p' \notin T' \wedge \text{vid}(t_i, q) < id \leq \text{vid}(t_l, q'))) \wedge \\
&(\nexists i \exists q : t_i = \mathbf{deliver}(q, m) \Rightarrow \exists b \exists id \exists D \exists T : a < b < c \wedge t_b = \mathbf{view}(p, id, D, T) \wedge \text{vid}(t_b, p) = \perp \wedge \\
&\ \exists k \exists id' \exists D' \exists T' : t_k = \mathbf{view}(q', id', D', T') \wedge p' \notin T' \wedge id \leq id' \leq \text{vid}(t_l, q'))
\end{aligned}
$$

3.14 Property (Agreed Messages)

1. *Agreed messages are causal messages. Formally:*

$$\text{agreed}(m) \equiv m.\text{type} \in \{ A, S \}$$

2. *If a process p delivers an agreed message m’, then after that event it will never deliver a message that has a lower ord value. Formally:*

$$t_a = \mathbf{deliver}(p, m) \wedge t_b = \mathbf{deliver}(p, m') \wedge \text{agreed}(m') \wedge \text{ord}(m) < \text{ord}(m') \Rightarrow a < b$$

3. *If a process p delivers an agreed message m’ before a* **trans_sig** *event in its current view, then p delivers every message with a lower ord value than m’ delivered in that view by any process. Formally:*

$$
\begin{aligned}
&t_c = \mathbf{deliver}(p, m') \wedge \text{agreed}(m') \wedge (\nexists b : b < c \wedge t_b = \mathbf{trans\_sig}(p) \wedge \text{vid}(t_b, p) = \text{vid}(t_c, p)) \Rightarrow \\
&\forall i \forall q \forall m : t_i = \mathbf{deliver}(q, m) \wedge \text{vid}(t_i, q) = \text{vid}(t_c, p) \wedge \text{ord}(m) < \text{ord}(m'); \exists a : t_a = \mathbf{deliver}(p, m)
\end{aligned}
$$

4. *If a process p delivers an agreed message m’ after a* **trans_sig** *event in its current view, then p delivers every message with a lower ord value than m’ sent by any processes in p’s next transitional set that were delivered in the same view as m’. Formally:*

$$
\begin{aligned}
&t_a = \mathbf{trans\_sig}(p) \wedge t_c = \mathbf{deliver}(p, m') \wedge t_d = \mathbf{view}(p, id', D', T') \wedge a < c < d \wedge \text{agreed}(m') \wedge \\
&\text{vid}(t_a, p) = \text{vid}(t_c, p) = \text{vid}(t_d, p) = id \Rightarrow \\
&\forall i \forall q \in T' \forall m \forall l \forall p' : t_i = \mathbf{send}(q, m) \wedge t_l = \mathbf{deliver}(p', m) \wedge \text{vid}(t_l, p') = id \wedge \text{ord}(m) < \text{ord}(m'); \\
&\exists b : t_b = \mathbf{deliver}(p, m)
\end{aligned}
$$

3.15 Property (Safe Messages)

1. *Safe messages are agreed messages. Formally:*

   $safe(m) \equiv m.type \in \{ S \}$

2. *If a process p delivers a safe message m before a* **trans_sig** *event in its current view id, then every member of that view delivers m, unless it crashes in id. Formally:*

   $t_a = \mathbf{view}(p, id, D, T) \wedge t_c = \mathbf{deliver}(p, m) \wedge safe(m) \wedge vid(t_c, p) = id \wedge$
   $\nexists b : a < b < c \wedge t_b = \mathbf{trans\_sig}(p) \Rightarrow$
   $\forall q \in D;\ \exists i \exists D' \exists T' \exists j : t_i = \mathbf{view}(q, id, D', T') \wedge (t_j = \mathbf{deliver}(q, m) \vee (t_j = \mathbf{crash}(q) \wedge vid(t_j, q) = id))$

3. *If a process p delivers a safe message m after a* **trans_sig** *event in its current view id, then every member of p's transitional set from p's next view delivers m, unless it crashes in id. Formally:*

   $t_a = \mathbf{view}(p, id, D, T) \wedge t_b = \mathbf{trans\_sig}(p) \wedge t_c = \mathbf{deliver}(p, m) \wedge t_d = \mathbf{view}(p, id'', D'', T'') \wedge$
   $safe(m) \wedge b < c \wedge vid(t_b, p) = vid(t_c, p) = vid(t_d, p) = id \Rightarrow$
   $\forall q \in T'';\ \exists i \exists D' \exists T' \exists j : t_i = \mathbf{view}(q, id, D', T') \wedge$
   $(t_j = \mathbf{deliver}(q, m) \vee (t_j = \mathbf{crash}(q) \wedge vid(t_j, q) = id))$

3.16 Property (Transitional Signals)

1. *At most one* **trans_sig** *event occurs at a process per view. Formally:*

   $t_a = \mathbf{trans\_sig}(p) \wedge vid(t_a, p) = id \Rightarrow$
   $\nexists b : b \neq a \wedge t_b = \mathbf{trans\_sig}(p) \wedge vid(t_b, p) = id$

2. *If two processes p and q are virtually synchronous (defined below) in a view id and p has a* **trans_sig** *event occur in that view, then q also has a* **trans_sig** *event occur in that view and they both deliver the same sets of agreed messages before and after their* **trans_sig** *events in that view. Formally:*

   $vsynchronous\_in(p, q, id) \wedge t_b = \mathbf{trans\_sig}(p) \wedge vid(t_b, p) = id \Rightarrow$
   $\exists j : t_j = \mathbf{trans\_sig}(q) \wedge vid(t_j, q) = id \wedge$
   $(\exists a \exists m : a < b \wedge t_i = \mathbf{deliver}(p, m) \wedge vid(t_a, p) = id \wedge agreed(m) \Leftrightarrow$
   $\exists i \exists m : i < j \wedge t_i = \mathbf{deliver}(q, m) \wedge vid(t_i, q) = id \wedge agreed(m)) \wedge$
   $(\exists c \exists m' : b < c \wedge t_c = \mathbf{deliver}(p, m') \wedge vid(t_c, p) = id \wedge agreed(m') \Leftrightarrow$
   $\exists k \exists m' : j < k \wedge t_k = \mathbf{deliver}(q, m') \wedge vid(t_k, q) = id \wedge agreed(m'))$

3.3 Extended Virtual Synchrony Model Extensions

This section presents the safety properties that the EVS model provides above and beyond the safety properties that the VS and EVS models share in common.

3.7 **Definition (EVS vsynchonous_in)** *If processes p and q both install the same view in the same previous view, then they were virtually synchronous in that previous view. Formally:*

$$\text{vsynchronous\_in}(p, q, id) \equiv \exists a \exists id' \exists D \exists T \exists i \exists D' \exists T' : \\ t_a = \mathbf{view}(p, id', D, T) \wedge t_i = \mathbf{view}(q, id', D', T') \wedge \\ \text{vid}(t_a, p) = \text{vid}(t_i, q) = id$$

3.17 **Property (EVS Transitional Set)**

1. *The transitional set for the first view installed at a process following a* **recover** *event is the empty set. Formally:*

   $t_a = \mathbf{view}(p, id, D, T) \wedge \text{vid}(t_a, p) = \perp \Rightarrow T = \varnothing$

2. *If a process p installs a view in a previous view, then the transitional set for the new view at p is a subset of the intersection between the two views' membership sets. Formally:*

   $t_a = \mathbf{view}(p, id, D, T) \wedge t_b = \mathbf{view}(p, id', D', T') \wedge \text{vid}(t_b, p) = id \Rightarrow T' \subseteq D \cap D'$

3. *If processes p and q install the same view, then q is included in p's transitional set for that view if and only if p's previous view was identical to q's previous view. Formally:*

   $t_a = \mathbf{view}(p, id'', D, T) \wedge \text{vid}(t_a, p) = id \wedge t_i = \mathbf{view}(q, id'', D', T') \wedge \text{vid}(t_i, q) = id' \Rightarrow$
   $(q \in T \Leftrightarrow id = id')$

4. *If processes p and q install the same view in the same previous view, then they have the same transitional sets in their new views. Formally:*

   $t_a = \mathbf{view}(p, id, D, T) \wedge t_i = \mathbf{view}(q, id, D', T') \wedge \text{vid}(t_a, p) = \text{vid}(t_i, q) \Rightarrow T = T'$

3.4 Virtual Synchrony Model Extensions

This section presents the extensions to the external signature and additional safety properties that the VS model provides beyond those that the VS and EVS models share in common.

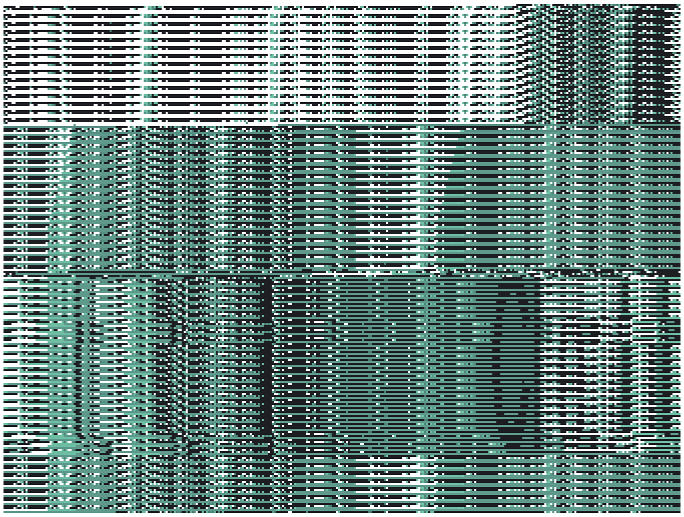


Figure 2: External signature of the VS Model.

Additional Interaction with the Client:

- input **flush**(p), $p \in P$
- output **flush_req**(p), $p \in P$

$$\textbf{Actions} := \textbf{Actions} \cup \{\ \textbf{flush}(p) \mid p \in P\ \} \cup \{\ \textbf{flush\_req}(p) \mid p \in P\ \}$$

3.8 **Re-Definition (VS pid)** *The pid of an event t_a is the process at which that event occurred. Formally:*

$$\begin{aligned} \text{pid}(t_a) := p \text{ if } & t_a = \textbf{trans\_sig}(p) \vee t_a = \textbf{crash}(p) \vee t_a = \textbf{recover}(p) \vee \\ & t_a = \textbf{flush}(p) \vee t_a = \textbf{flush\_req}(p) \vee (\exists m : t_a = \textbf{deliver}(p, m) \vee t_a = \textbf{send}(p, m)) \vee \\ & (\exists id \exists D \exists T : t_a = \textbf{view}(p, id, D, T)) \end{aligned}$$

3.9 **Definition (VS vsynchonous_in)** *If processes p and q both install a view in the same previous view, id, and q is in p's transitional set, then they are virtually synchronous in id*[3]*. Formally:*

$$\begin{aligned} \text{vsynchronous\_in}(p, q, id) \equiv\ & \exists a \exists id' \exists D \exists T \exists i \exists D' \exists T' : \\ & t_a = \textbf{view}(p, id', D, T) \wedge t_i = \textbf{view}(q, id', D', T') \wedge \\ & \text{vid}(t_a, p) = \text{vid}(t_i, q) = id \wedge q \in T \end{aligned}$$

[3] The following definition of the VS transitional set makes VS vsynchronous_in a reflexive, symmetric and transitive relation on processes in a view.

3.18 **Property (VS Initial View Event)** *Every* **flush, flush_req, send, deliver,** *and* **trans_sig** *event occurs within some view. Formally:*

$$t_a = \textbf{flush}(p) \lor t_a = \textbf{flush\_req}(p) \lor t_a = \textbf{send}(p, m) \lor t_a = \textbf{deliver}(p, m) \lor t_a = \textbf{trans\_sig}(p) \Rightarrow \mathrm{vid}(t_a, p) \neq \bot$$

3.19 **Property (Sending View Delivery)** *Messages are delivered in the view in which they are sent. Formally:*

$$t_a = \textbf{deliver}(p, m) \land \mathrm{vid}(t_a, p) = id \Rightarrow \exists i \exists q : t_i = \textbf{send}(q, m) \land \mathrm{vid}(t_i, q) = id$$

3.20 **Property (Flush Requests and Flushes)**

1. *At most one* **flush_req** *event occurs in a view at a process. Formally:*

   $$t_a = \textbf{flush\_req}(p) \Rightarrow \nexists b : b \neq a \land t_b = \textbf{flush\_req}(p) \land \mathrm{vid}(t_a, p) = \mathrm{vid}(t_b, p)$$

2. *At most one* **flush** *event occurs in a view at a process. A* **flush** *event is preceded by a* **flush_req** *event in that view at that process. No* **send** *events follow a* **flush** *event in a view at a process. Formally:*

   $$t_b = \textbf{flush}(p) \Rightarrow \nexists d : d \neq b \land t_d = \textbf{flush}(p) \land \mathrm{vid}(t_b, p) = \mathrm{vid}(t_d, p) \land$$
   $$\exists a \nexists c \exists m : a < b < c \land t_a = \textbf{flush\_req}(p) \land t_c = \textbf{send}(p, m) \land \mathrm{vid}(t_a, p) = \mathrm{vid}(t_b, p) = \mathrm{vid}(t_c, p)$$

3. *Every* **view** *event, except for the first following a* **recover** *event, at a process is preceded by a* **flush** *event. Formally:*

   $$t_b = \textbf{view}(p, id, D, T) \Rightarrow \exists a : a < b \land (t_a = \textbf{flush}(p) \lor t_a = \textbf{recover}(p)) \land \mathrm{vid}(t_a, p) = \mathrm{vid}(t_b, p)$$

3.21 **Property (VS Transitional Set)**

1. *The transitional set for the first view installed at a process following a* **recover** *event is the empty set. Formally:*

   $$t_a = \textbf{view}(p, id, D, T) \land \mathrm{vid}(t_a, p) = \bot \Rightarrow T = \emptyset$$

2. *If a process p installs a view in a previous view, then the transitional set for the new view at p is the union of p with a subset of the intersection between the two views' membership sets. Formally:*

   $$t_a = \textbf{view}(p, id, D, T) \land t_b = \textbf{view}(p, id', D', T') \land \mathrm{vid}(t_b, p) = id \Rightarrow p \in T' \land T' \subseteq D \cap D'$$

3. *If processes p and q install the same view and q is included in p's transitional set for that view, then p's previous view was identical to q's previous view. Formally:*

   $$t_a = \textbf{view}(p, id', D, T) \land \mathrm{vid}(t_a, p) = id \land t_i = \textbf{view}(q, id', D', T') \land q \in T \Rightarrow \mathrm{vid}(t_i, q) = id$$

4. *If processes p and q install the same view and q is included in p's transitional set for that view, then p and q have the same transitional sets for that view. Formally:*

   $$t_a = \textbf{view}(p, id', D, T) \land t_i = \textbf{view}(q, id', D', T') \land q \in T \Rightarrow T = T'$$

4 VS Algorithm Design

This section describes some of the problems involved with implementing VS on top of EVS. It also describes, in general terms, the algorithms that this work developed to solve those problems.

4.1 Differences between EVS and VS

The main difference between the EVS and VS models is that the VS model offers Sending View Delivery semantics while EVS only offers Same View Delivery semantics.

This difference in message delivery semantics is responsible for the differences in the models external signatures. Friedman and van Renesse have shown [FvR95] that in order to maintain Sending View Delivery without violat ing other useful safety properties, such as Self-Delivery and Virtual Synchrony, clients must not be allowed to send messages for a certain period of time before each VS view is installed. Therefore, a VS implementation must signal its clients that an underlying membership change has occurred and request that the client stop sending messages in its current view, so that this new view information can be delivered. The client, when it is ready, responds to this request with a flush message that closes its current view. After closing its previous view a client is not allowed to send more messages until a new view is installed. Thus, to enable Sending View Delivery for a client, the VS model has the additional flush and flush request events that the EVS model does not.

EVS systems never request authorization from clients in order to install new views; they simply determine and install new views as they deliver messages while maintaining the appropriate properties. This makes the EVS model uniquely suited for a client-daemon implementation. Most GCSs are implemented such that each client process acts a GCS process, meaning that the client process itself, or a subcomponent of the client process ensures that the safety and liveness properties of the system are maintained. In a client -daemon system, a set of dedicated, long-lived daemon processes is responsible for maintaining the safety and liveness properties of the system. Client processes connect to one of the daemons and send and receive messages through that daemon. That daemon acts as a representative for that client in the group communication system, ensuring all of the safety and liveness properties of the model.

This client -daemon architecture has many performance advantages over client-based architectures. First, almost every variable algorithmic cost in a GCS is tied to how many processes are involved in a procedure. In practice, the number of daemons is small relative to the number of client processes. This means that almost every algorithmic cost is drastically reduced in this architecture. Second, in this kind of system, client process membership changes can be implemented with a single Agreed message [4] [Spread]. This kind of membership change is far less expensive than membership changes in client based systems, which often consist of multiple n-to-n communication rounds during

[4] In this architecture, Same View Delivery is not maintained in certain rare scenarios.

which messages cannot be sent. The client-daemon architecture also has these more expensive membership changes, however, these heavyweight membership changes only occur due to daemon membership changes. Daemon membership changes only occur when particular network components partition or merge and daemons crash or recover. In most environments, these are relatively rare events compared to client processes joining or leaving the group communication system. The high performance of this client-daemon EVS architecture, along with its inexpensive and non-blocking client level memberships are what make it a prime candidate on top of which to implement a client level VS system.

4.2 Problem Description: Maintaining Safety and Liveness Properties

As mentioned in the previous major section, a GCS consists of a set of safety and liveness properties. In the previous section, the safety properties of the EVS and VS systems were laid out in detail. This section informally discusses the liveness properties that must also be maintained to correctly implement a GCS model. This section also points out some of the constraints that maintaining these different properties enforces on any VS algorithm and comments on how they apply, in particular, to implementing VS on top of EVS.

4.2.1 Membership Liveness Properties

The main purpose of a GCS is to allow its clients to communicate with other clients connected to the system. To be truly useful, most GCSs maintain a membership liveness property that requires a certain level of precision from their membership service. The main point of this liveness property is to require a GCS's membership service to eventually reflect the underlying communication connectivity of its clients. This GCS liveness property is difficult to formulate [ACBMT95] and has been proven impossible to maintain in every situation [CHTCB96]. Roughly speaking, this liveness property requires that if an underlying connectivity change occurs and the new connectivity exists forever, then the new connectivity must eventually be uniformly and accurately presented to the connected clients.

In order to accomplish this feat, any VS algorithm must use some form of distributed agreement to get all of the members of the underlying stable membership to agree upon and install the same VS view. For the purposes of this work, maintaining this property requires that if an EVS view persists forever, that eventually the VS algorithm will install the **same** VS view to all of the members of the underlying view and that it will not install any further views.

If an algorithm installs a view that does not match the underlying system's view (as the algorithm knows it) at the time of installation, then the installed view is said to be obsolete. If an algorithm installs an obsolete view it will eventually have to re-execute to install a more accurate view. An algorithm that installs obsolete views will usually generate more views than one that does not, which often causes client processes to do unnecessary state synchronization work. Therefore, algorithms that generate obsolete views are generally considered less desirable than those that do not.

4.2.2 Membership Safety Properties

The safety properties that the membership service must maintain are relatively few. In particular, it must maintain the Self-Inclusion, Membership Agreement and Local Monotonicity properties.

A VS algorithm can take two approaches to maintaining these properties. These properties can either be provided by a separate membership service, or the algorithm itself can use a form of distributed agreement to agree upon view identifiers and membership sets.

4.2.3 Multicast Liveness Properties

The other common liveness properties of GCSs concern the eventual delivery of messages in stable views. If the connectivity of the underlying communication system eventually stabilizes forever, then, as described above, an algorithm built on top of such a system must eventually reflect that connectivity to its clients in a stable view. Furthermore, messages that are sent in such a stable view must eventually be delivered to all the members of that view.

For the purposes of this work, this means that a VS algorithm cannot drop messages in a stable view. Of course, there is no practical way for an algorithm to determine if a stable view has been reached. Therefore, if the algorithm installs a view and no further underlying changes have occurred yet, then messages received from the underlying system must eventually be delivered to all of the members of that view.

4.2.4 Multicast Safety Properties

There are a host of safety properties on the multicast service. Most of these properties concern the ordering and reliability of different message types. Many algorithms have been developed that maintain the safety properties of the respective message types.

A VS algorithm must either implement some of these algorithms itself, or it can have many of those services done for it by a separate multicast service. In this latter case, the algorithm simply must ensure that the underlying service, along with injected alterations such as inserting, dropping or reordering messages maintains the correct safety properties.

4.3 Single Round VS Algorithm Using FIFO Messages

This section discusses the particular algorithm that this work developed to implement VS on top of EVS.

4.3.1 VS Algorithm Design Philosophy

The main thrust of this work was to develop an algorithm that maintained the safety and liveness properties provided by the EVS system below it and with minimal additional work added the additional safety properties of the VS model. In effect, the algorithm should “interfere” with the operation of the EVS system just enough to implement the extra safety semantics of the VS model. If done correctly, the liveness properties of the underlying EVS layer will be implicitly maintained and many of the EVS system's safety properties will "bleed through" as well. This leveraging of GCS properties is the reason why the algorithms implemented in this manner can be so much simpler than "native" algorithms.

First and foremost, the VS algorithm will exploit the EVS membership service as much as possible. It does this by using the views provided by the EVS system as potential VS views to be installed. This heuristic will almost implicitly maintain the safety properties of the underlying membership service. Furthermore, if new VS views are only installed in response to EVS views being installed, and the algorithm eventually installs the most recent EVS view, then the liveness properties of the EVS membership service will implicitly be maintained as well.

Second, the VS algorithm will implicitly maintain many of the safety properties of the underlying multicast service by only performing minimal reordering of messages before they are either dropped or delivered. Obviously, the basic multicast safety properties such as Property (No Duplication) and Property (Delivery Integrity) are easily maintained.

4.3.2 Algorithm Description

The VS algorithm's presentation is broken into two sections. The first discusses the membership portion of the algorithm that installs VS views. The second discusses the message delivery portion of the algorithm.

4.3.2.1 VS View Installation

The core concept of the VS membership algorithm this work developed is quite simple. When a client process attempts to install a VS view, it attempts to install a view that matches its most recent EVS view. It does this by first multicasting a FIFO "flush message" marked with the view identifier of that EVS view. The client then tries to collect a flush message marked with that view identifier from each of the members of that EVS view. If it achieves this, then it installs that VS view. If before the client can collect all of the necessary messages another EVS view is installed at the client, then it abandons the previous view and tries to install a new VS view that matches its new most recent EVS view.

A client tries to install a new VS view only in two cases: (1) upon startup/recovery and (2) when an EVS view is received while the process is in an established VS view. In this second case, the algorithm notifies the client that the underlying connectivity has changed by generating a flush request event. When the client is ready, it responds with a flush event that authorizes the client's algorithm to try and actively install the next VS view. The algorithm then generates its first flush message and proceeds as above described. While actively trying to install the next VS view a client is blocked from sending messages.

It is easy to see how this heuristic for installing VS views maintains both the safety and liveness properties of the EVS membership service. The safety properties are almost trivially maintained, because the VS algorithm uses EVS views as VS views in the order that they are installed at clients. The membership liveness property is maintained, because if an EVS view is installed that is the stable EVS view, then once all of the member processes flush[5] their previous VS views those flush messages will be delivered to all of the other members. Therefore, all of the members of the stable EVS view will install the VS view that matches the stable EVS view. Also, because the stable EVS view is the last view ever installed by the EVS system, this algorithm will not attempt to install any further VS views.

4.3.2.2 VS Message Delivery

The core concept of the VS multicast algorithm this work developed is also fairly simple. All VS messages are marked with the VS view in which they are sent. A process also maintains a set of processes, called Vs_Survivors, from its most recent VS view, if any, that have been in the transitional set of every EVS view installed since the process' most recent VS view. The name of this set is apt because it is the set of processes that the EVS system hypothesizes have been virtually synchronous with this process since it installed its last VS view.

If a process receives a message from the EVS system that is marked with the view identifier of its most recent EVS view and that identifier is different than its most recent VS view's identifier, then it buffers the message. If another EVS view is installed at the process before it receives all the necessary flush messages to install its most recent EVS view, then any buffered messages are dropped. When a process installs a VS view it then delivers any messages that it currently has buffered. If a process receives a message marked with its current VS view identifier, then if the sender is in the process' Vs_Survivors set, it immediately delivers the message. In all other cases, all messages are dropped.

It is more difficult to understand the reasoning behind these message delivery heuristics than the membership installation heuristics. The buffering of messages is fairly easy to understand. If a process' most recent EVS view is different than its most recent VS view, then that process' next VS view could be its current EVS view. Therefore, if it receives messages that were sent in that VS view it should buffer them, in case this process installs that VS view. If it does not install that VS view, then by EVS Property (Local Monotonicity), the algorithm will never install that view and can safely drop those messages.

The buffering the algorithm performs does not reorder messages from the point of view of the VS system. The only way the algorithm could violate the ordering properties provided by the EVS system is if it buffered a message m and then later received and delivered a message m' that was ordered after m, before it delivered m. A process never delivers any message that does not match its VS or EVS view identifier upon receipt and it buffers every message marked with its EVS view identifier when that identifier is different than its VS identifier. If after buffering some messages a process receives a

[5] Note, that if a member never authorizes closing its previous VS view, then it is completely legal (i.e. – not a violation of the liveness property) to never install the stable VS view. Not installing the stable VS view is due to a client willfully blocking the system and is not due to any deficiency of the algorithm.

message m for its current VS view, then it is legal to deliver m immediately. This is because, as is shown below in the proof of Lemma 5.7 (VS Message Ordering), a process only buffers non-Causal messages. Therefore, if the process immediately delivers m in its current VS view, then the only ordering violation that could be violated is the FIFO message ordering constraint. But this violation cannot occur because in order for the sender to have installed a new VS view (it did because the message in the queue was marked with the process' current EVS view identifier) it must have sent a FIFO flush message for the view it installed after sending m. Therefore, because the process just received m, due to the FIFO ordering requirement, no FIFO message sent after m could have been received by this process yet. Any messages from the sender that this process already buffered must be Reliable messages it sent in a following VS view. Therefore, delivering m immediately while messages have been buffered cannot violate the FIFO ordering property. Therefore, the algorithm maintains all of the ordering properties given by the underlying EVS system.

The only other way the algorithm delivers messages is if a message is marked with the process' most recent VS view identifier and the message's sender is in its Vs_Survivors set. The reasoning behind this heuristic is that the EVS message reliability (no-holes) safety properties span multiple EVS views for messages it delivers from processes that have been in the transitional sets of each of those EVS views. For senders not in all of those transitional sets, the reliability guarantees may or may not hold across all of those EVS views. Therefore, it is only correct to deliver messages in a VS view if the sender has been in the transitional sets of all the EVS views that followed the EVS view corresponding to their current VS view. If this policy was not followed then this algorithm could introduce a causal hole in the stream of messages in the VS view at this process.

In all other cases, messages are dropped as they are received from the EVS system. These cases are the result of: (1) messages sent in VS views that the process will never install and (2) messages sent in VS views that the process previously installed, but has since installed another VS view. By Property (Sane View Delivery) the identifier of the EVS delivery view of a message is greater than or equal to the identifier of the EVS view in which the message was sent. Therefore, because the identifier on the message does not match the process' current EVS view identifier it must be less than the process' current EVS view identifier.

If the message was sent in a VS view that the process had not previously installed, then the process will never install that VS view. This is because the algorithm always tries to install a VS view matching its most recent EVS view and the message's VS view identifier is less than the process' most recent EVS view identifier.

If the message was sent in a VS view that the process had previously installed, then the receiver must have partitioned away from the sender at some point and installed a VS view without the sender in the membership set. This is because, if the receiver had not partitioned away at some point, the sender would have been in the membership set of every following EVS view installed at the receiver up until it received the message in question. Therefore, in order to install another VS view, the recipient would be required to receive a flush message from the sender. Recall, that once a process generates a flush message in a VS view it can no longer send any messages. Therefore, because flush messages are FIFO messages and the recipient installed a following VS view, the sender must not have been in a membership set of at least one of the VS views installed at the recipient.

These heuristics for message delivery also maintain the necessary message delivery liveness properties. Recall, that if a stable underlying view is reached, then all the members of that view must install the same VS view and all the members of that view must deliver any messages sent in that view. It was shown above that once a stable EVS view is reached and all the members of that view close their previous VS view, that all the members of that EVS view would install the corresponding VS view. When they install that view, each member's Vs_Survivors set contains all of the members of that view. Therefore, since there are no more following EVS views, Vs_Survivors will always contain the sender of any message delivered in that EVS view, by EVS Delivery Integrity. Therefore, the algorithm will always deliver messages sent in the stable VS view and the multicast liveness property is implicitly maintained. If any messages are sent in the stable VS view that are received in the corresponding EVS view before a process installs the stable VS view, then the receiving process will buffer those messages and deliver them upon installing the stable VS view.

4.3.2.3 VS Transitional Sets and Signals

The only other non -trivial properties that the algorithm must maintain are th e safety properties of the transitional sets and signals.

When the EVS system generates a transitional signal in an EVS view, the algorithm generates a transitional signal if it subsequently delivers an agreed message or an EVS view event occurs in its V S view. The algorithm also generates a transitional signal in a VS view if an EVS view event removes a member from its Vs_Survivors set. The algorithm generates at most one transitional signal per VS view. By delivering the transitional signal immediately before an Agreed message is delivered or when an EVS view event occurs, processes that are virtually synchronous through VS views will deliver the transitional signal at the same point in the stream of agreed messages in that VS view.

The transitional set of a VS view is simply the process' Vs_Survivors set when it installs a VS view. This is because the members of a process' Vs_Survivors set were virtually synchronous to it through all the EVS views since the last one corresponding to their previous VS view up to the EVS view corresponding to the process' following VS view. If a process receives a flush message from a process q that is one of the members of its Vs_Survivors set, then it received and delivered all the messages that q sent in their previous view. As is proved later, such members will have the same Vs_Survivors sets and if they both install the same following VS view they both delivered the same set of messages in their previous VS view.

4.4 Algorithm Evaluation

The previous subsection laid out the VS algorithm this work developed and informally showed how it maintains the important safety and liveness properties of the VS model. This section attempts to evaluate the good and bad attributes of this algorithm, in terms of its algorithmic overhead and the quality of the semantics that it offers.

4.4.1 VS View Installation Heuristics

This algorithm is very responsive to its underlying layer’s membership service. Once the lower layer installs a view, the VS algorithm immediately abandons the VS view it was trying to install and attempts to install a VS view matching the current client connectivity. In fact, the only time the algorithm installs an obsolete view is when it installs a VS view after it receives an EVS transitional signal in its current EVS view. In this case, the transitional signal indicates that a new view is about to be installed, so the algorithm has knowledge of an impending view change. However, the algorithm is designed to work with FIFO flush messages and transitional signals have no guaranteed ordering with respect to these messages. Therefore, the algorithm cannot use transitional signals to decide on whether or not to install VS views. If this algorithm did, then several processes might install a VS view while others might not, which is a very undesirable situation. Regardless, this situation occurs very rarely in practice.

This membership algorithm’s performance is competitive with other client-based VS algorithm implementations. Any VS algorithm effectively has to conduct an n-to-n round of communication among the potential members to agree on views to install and to close previous views. This n-to-n round of communication can be done hierarchically to reduce the number of messages generated [ACDV97], but each process must authorize installing the view and each process must receive some form of agreement from each of the other potential members of the view. The algorithm described here uses inexpensive FIFO messages for its one round of agreement. In order to “flush out” messages sent in the previous view and to close those views, any VS algorithm will effectively use at least FIFO messages for its round(s) of communication.

The additional overhead that this algorithm pays beyond what most other algorithms would pay is the time it takes for the EVS system to install its views. As was discussed previously, lightweight client membership changes can be implemented in a non-blocking manner using a single Agreed message sent amongst the daemons. Therefore, this algorithm pays the additional cost of first waiting for a sent message to become stable amongst the daemon processes, while other implementations could begin working on installing a new view immediately. This additional cost effectively translates to receiving an acknowledgement of receipt of the message from each of the daemons in a configuration. This additional latency depends on the number of daemons and the configuration of the network between the daemons.

In the case of a heavyweight daemon membership changes, this algorithm performs poorly compared to other implementations of VS algorithms. In this case, this algorithm must wait for the heavyweight daemon membership to be established before it tries to install a corresponding VS view. A heavyweight membership algorithm usually consists of at least one, if not multiple, n-to-n round(s) of communication between the daemons. The additional overhead that the algorithm pays in this case is the time for those rounds of communication to complete. This additional latency depends on the number of daemons, the configuration and stability of the network and the complexity of the synchronization algorithm that the daemons use to establish and close their heavyweight views.

4.4.2 VS Message Delivery Heuristics

The algorithm effectively delivers messages as it receives them from the EVS system. Therefore, it directly inherits the high-performance characteristics of the client-daemon EVS model for message delivery. In fact, the only additional latency that the algorithm adds to message delivery is for when it buffers non-Causal messages that were received too early to deliver. Of course, any VS algorithm would have to do this kind of buffering – almost any algorithm would have to buffer or drop a message it received for a potential VS view it had not yet installed. The only costs that this algorithm pays that others would not pay is that the EVS system maintains its ordering and reliability guarantees for messages that may not be pertinent to the VS ordering and reliability guarantees. For example, when two network components merge together, processes in multiple VS views are all potentially sending messages. These messages sent in different views are not really dependent upon one another from the VS system's point of view, but the EVS system will still maintain its safety properties and will introduce causal dependencies based on send and deliver events in the new EVS view. Therefore, the EVS layer may generate some unneeded overhead as it seeks to maintain safety properties that are unnecessary from the VS system's point of view.

This algorithm performs no message recovery and drops messages from live and connected components. Normally, these attributes would be considered fatal flaws in a group communication algorithm. However, the underlying EVS system already performs powerful message recovery for the VS algorithm. The expense of adding additional message recovery on top of the EVS's might, in fact, not be worth the potential benefit. Additionally, any VS algorithm will occasionally need to drop messages from connected clients, even when those messages are sent in VS views that this process previously installed. This observation is best-illustrated by example:

> Client processes p and q have both installed the same VS view id, of which they are the only members. An underlying view change partitions p and q away from each other into lower level singleton views. The client's VS algorithms dutifully deliver notification that the underlying client connectivity has changed to p and q. Process p flushes its view and installs a new singleton VS view id', while process q ignores its signal and continues sending messages in id. Another underlying view change then merges p and q back together, while q continues sending messages in id. The VS algorithm cannot deliver any of q's new messages to p because they were sent in id and p has already installed a later view id'.
>
> Client processes p and q have both installed the same VS view id, of which they are the only members. An underlying view change partitions p and q away from each other into singleton views. The client's VS algorithms dutifully deliver notification that the

> underlying client connectivity has changed to p and q. Neither process flushes its current VS view and continues to send messages. Another underlying view change merges p and q back together. If the VS algorithm buffered the messages that p and q sent while they were partitioned, then a complex algorithm could recover and deliver some of the Causal and weaker service messages that they sent.

This second example is a little overly optimistic about message recovery. A process can remain in a VS view for an arbitrarily long period of time, although this does not usually happen. Therefore, a process cannot buffer all the messages sent by the process in a view unless, theoretically, it has infinite memory. Furthermore, if any Agreed messages were sent and delivered by the processes while they were partitioned, the other process will be unable to deliver these messages due to the Agreed ordering requirement. After the first such message the other process will not be able to recover even subsequently sent FIFO messages. This example demonstrates that message recovery and re-synchronization is only really useful to capture and correct short-term network instabilities. In that case, the message recovery that the EVS system performs should be just as effective as a VS algorithm's even though it will only attempt to recover messages over a potentially shorter "message horizon."

By making this VS algorithm attempt to perform message recovery the clean separation that previously existed between the VS algorithm and the EVS system would be ruined. The VS algorithm would either have to run its own message ordering and recovery algorithms on top of the underlying system's , or understand and manipulate the underlying layer's message delivery subsystem. Either of these options would completely destroy the simplicity of the algorithm to achieve very questionable benefit.

One last interesting point about this algorithm's behavior – because of the heuristics it uses, the usual test of whether or not two processes were virtually synchronous in a view cannot be used. In almost every GCS specification, two processes were virtually synchronous in a view id if they both installed the same new view in id. To maintain this property, an algorithm has to perform message recovery and/or potentially install obsolete views. This point is demonstrated by the second example above. In that example, after the two processes remerged if they attempted to install a new VS view and there were messages delivered in their previous view that the other could not deliver, then the processes would have to install an obsolete view. This is because, otherwise, they would install the same new view in the same previous view. If they did that, Property (Virtual Synchrony) would require them to deliver the same set of messages in the former view. But as was just shown, this is not possible if they deliver certain types of messages while partitioned from one another. Of course, using this same example, if the processes did not deliver Agreed messages while partitioned, then they could recover the messages that they delivered while partitioned. In this case, the processes could either potentially continue in their previous view as if nothing had happened, or they could install a new view. However, as pointed out above, this solution theoretically requires a process to potentially buffer all of the messages that it sends in a view.

Instead of dealing with these problems, this work changed the test for virtual synchrony to not only require the processes to install the same view in their same previous view, but to also require those processes to be in each others transitional sets. If they are not in each other's trans itional sets, then the fact that they installed the same view in their same previous view <u>does not</u> imply that they were virtually synchronous in their previous view. This model is actually no weaker than the original specification of virtual synchrony. It does allow the GCS to use more trivial solutions, but a good implementation can maintain the same strength of semantics as the original virtual synchrony property. The only other difference this change makes is that applications that depend on using view identifiers to determine if two processes were virtually synchronous or not in a view must now also remember their transitional sets for those views. Changing the model this way allows processes not to install obsolete views while also not requiring them to do message recovery.

5 Single Round VS Algorithm Pseudo-code

This section presents an event driven pseudo-code algorithm for implementing a VS system on top of an EVS system from the viewpoint of a single client or GCS process. The EVS system generates events that the algorithm intercepts and handles. This algorithm then operates and generates client visible VS events. In effect, the trace of events at a GCS automaton has both EVS and VS events intermingled, but a client of the system will only see the VS events.

In this presentation, events generated by the EVS system have **evs_** as a prefix and VS events generated by the algorithm have **vs_** as a prefix. The two events, request_flush and request_send are generated by the client process and do not directly generate VS events as the corresponding events can only be generated legally under certain preconditions.

```
Process_Variables :=
    { Vs_id | Vs_id $\in$ (VID $\cup$ {$\perp$}) } $\cup$
    { Vs_Survivors | Vs_Survivors $\in$ $2^P$ } $\cup$
    { Vs_Flushers | Vs_Flushers $\in$ $2^P$ } $\cup$
    { Vs_delivd_trans_sig | Vs_delivd_trans_sig $\in$ B } $\cup$
    { Vs_delivd_flush_req | Vs_delivd_flush_req $\in$ B } $\cup$
    { Vs_sent_flush | Vs_sent_flush $\in$ B } $\cup$
    { Evs_id | Evs_id $\in$ (VID $\cup$ {$\perp$}) } $\cup$
    { Evs_Members | Evs_Members $\in$ $2^P$ } $\cup$
    { Evs_delivd_trans_sig | Evs_delivd_trans_sig $\in$ B } $\cup$
    { Delay_Queue }

case Event is

recover:
    Vs_id := $\perp$                   // identifier of most recent VS view
    Vs_Survivors := $\emptyset$            // tracking set for VS transitional set and msg delivery
    Vs_Flushers := $\emptyset$             // members that have flushed EVS view Evs_id
    Vs_delivd_trans_sig := true    // has a transitional signal been delivered in the current VS view?
    Vs_delivd_flush_req := true    // has a flush request been delivered in the current VS view?
    Vs_sent_flush := true          // has a flush message been sent in the current VS view?
    Evs_id := $\perp$                  // identifier of most recent EVS view
    Evs_Members := $\emptyset$             // membership set of most recent EVS view
    Evs_delivd_trans_sig := false  // has a transitional signal been delivered in the current EVS view?
    Delay_Queue := new Queue()     // a FIFO message queue

evs_trans_sig:
    Evs_delivd_trans_sig := true

request_flush:
    if (Vs_delivd_flush_req $\wedge$ $\neg$Vs_sent_flush)
        // distinguishes flush msgs from all other msgs and marks with the current EVS view id
        vs_flush := evs_send(new flush_msg(Evs_id))
        Vs_sent_flush := true
    else
        illegal, generate user error
```

```
request_send(m):
    if ($\neg$Vs_sent_flush)
        // marks every sent message with the id of the VS sending view
        vs_send(m) := evs_send(new msg(m, Vs_id))
    else
        illegal, generate client error

evs_view(id, D, T):
    if ($\neg$Vs_delivd_flush_req)
        vs_flush_req
        Vs_delivd_flush_req := true

    else if (Vs_sent_flush)
        // distinguishes flush msgs from all other msgs and marks it with the current EVS view id
        evs_send(new flush_msg(id))

    if ($\neg$Vs_delivd_trans_sig $\wedge$ (Evs_delivd_trans_sig $\vee$ (Vs_Survivors $\cap$ T $\subset$ Vs_Survivors)))
        vs_trans_sig
        Vs_delivd_trans_sig := true

    Vs_Survivors := Vs_Survivors $\cap$ T
    Vs_Flushers := $\emptyset$
    Evs_id := id
    Evs_Members := D
    Evs_delivd_trans_sig := false
    Delay_Queue.clear()                    // drops any messages that were pushed onto Delay_Queue

evs_deliver(m):
    if (is_flush_msg(m))
        if (flush_memb_id(m) = Evs_id)
            Vs_Flushers := Vs_Flushers $\cup$ m.sender
            if (Vs_Flushers = Evs_Members)
                vs_view(Evs_id, Evs_Members, Vs_Survivors)
                Vs_id := Evs_id
                Vs_Survivors := Evs_Members
                Vs_Flushers := $\emptyset$
                Vs_delivd_trans_sig := false
                Vs_delivd_flush_req := false
                Vs_sent_flush := false
                while ($\neg$Delay_Queue.empty())
                    vs_deliver(Delay_Queue.pop_head())

    else if (msg_vs_id(m) = Vs_id)             // gets the id of the VS view in which m was sent
        if (m.sender $\in$ Vs_Survivors)
            if (agreed(m) $\wedge$ $\neg$Vs_delivd_trans_sig $\wedge$ Evs_delivd_trans_sig)
                vs_trans_sig
                Vs_delivd_trans_sig := true
            vs_deliver(reg_mess(m))            // reg_mess removes the marked sending VS view id

    else if (group_id(m) = Evs_id)
        Delay_Queue.push_tail(reg_mess(m))     // reg_mess removes the marked sending VS view id

    // if m is not vs_delivered or pushed onto Delay_Queue then it is dropped
```

6 Proof of Correctness

This section formally proves that the previously presented algorithm maintains all of the safety properties of the VS model. Section 3 informally discussed how the algorithm maintains the liveness properties of the VS model.

6.1 **Definition (Variable Function)** *A variable function is a function with the name of a process variable that takes a process and an event as parameters and returns the value of the process variable exactly when that event occurred at that process. If the variable is undefined at that point in the trace for that process, then the variable function is undefined as well. Formally:*

$$\mathrm{var}(t_i, p) := \begin{cases} \text{(value of var at p at } t_i\text{)} & \text{if } \mathrm{pid}(t_i) = p \wedge t_i \neq \textbf{crash}(p) \wedge t_j \neq \textbf{recover}(p) \\ \text{undefined} & \text{otherwise} \end{cases}$$

6.1 **Lemma (Evs_id***)* *Evs_id at a process is equal to the view identifier of the most recent* **evs_view** *event at that process, or* ⊥ *if no* **evs_view** *event has occurred at that process since its most recent* **recover** *event. Formally:*

$$\begin{aligned}
&\mathrm{pid}(t_l) = p \wedge t_l \neq \textbf{crash}(p) \wedge t_l \neq \textbf{recover}(p) \wedge \\
&(\exists a \nexists b \exists id \exists D \exists T : a < b < l \wedge t_a = \textbf{recover}(p) \wedge t_b = \textbf{evs\_view}(p, id, D, T) \Leftrightarrow \mathrm{Evs\_id}(t_l, p) = \perp) \wedge \\
&(\exists i \nexists j \exists id' \exists D' \exists T' : i < j < l \wedge t_i = \textbf{evs\_view}(p, id', D', T') \wedge \\
&\ (t_j = \textbf{recover}(p) \vee \exists id'' \exists D'' \exists T'' : t_j = \textbf{evs\_view}(p, id'', D'', T'')) \Leftrightarrow \mathrm{Evs\_id}(t_l, p) = id' \wedge id' \neq \perp)
\end{aligned}$$

$$\begin{aligned}
&(\mathrm{Evs\_id}(t_l, p) = \perp \Leftrightarrow \mathrm{Evs\_Members}(t_l, p) = \emptyset) \wedge \\
&(\mathrm{Evs\_id}(t_l, p) = id \wedge id \neq \perp \wedge \mathrm{Evs\_Members}(t_l, p) = D \Rightarrow \exists a \exists T : a < l \wedge t_a = \textbf{evs\_view}(p, id, D, T))
\end{aligned}$$

Proof: From the algorithm, after a **recover** event occurs Evs_id and Evs_Members are immediately set, respectively, to ⊥ and ∅. After an **evs_view** event occurs, Evs_id and Evs_Members are immediately set, respectively, to the identifier and membership set of that EVS view. There are no other cases under which either Evs_id or Evs_Members is modified. By the definition of the **evs_view** action, the identifier of an EVS view is never ⊥ and the membership set of an **evs_view** event is never ∅, due to EVS Property 2.2 (Self-Inclusion).

6.2 **Lemma (VS Views)** *If a process installs a VS view, then the identifier and membership set of that view are, respectively, the view identifier and membership set of the most recent* **evs_view** *event at that process. Formally:*

$$\begin{aligned}
&t_c = \textbf{vs\_view}(p, id, D, T) \Rightarrow \exists a \nexists b \exists T' : a < b < c \wedge t_a = \textbf{evs\_view}(p, id, D, T') \wedge \\
&(\exists id' \exists D' \exists T'' : t_b = \textbf{evs\_view}(p, id', D', T''))
\end{aligned}$$

Proof: By the algorithm, **vs_view** events only occur immediately in response to an **evs_deliver** event. Furthermore, the view identifier and membership set of a VS view are, respectively, Evs_id and Evs_Members at the time of the instigating **evs_deliver** event. By EVS Property 2.1 (Initial View Event) every **evs_deliver** event at a process occurs within some EVS view at that process. Therefore, by Lemma 5.1 (Evs_id) when a

vs_view event occurs at a process Evs_id and Evs_Members are, respectively, equal to the most recent **evs_view**’s identifier and membership set at that process.

6.3 **Lemma (Vs_id)** *Vs_id is equal to the view identifier of the most recent* **vs_view** *event at the process or ⊥ if no* **vs_view** *event has occurred at the process since the most recent* **recover** *event. Formally:*

$\mathrm{pid}(t_l) = p \wedge t_l \neq \textbf{crash}(p) \wedge t_l \neq \textbf{recover}(p) \wedge$
$(\exists a \nexists b \exists id \exists D \exists T : a < b < l \wedge t_a = \textbf{recover}(p) \wedge t_b = \textbf{vs\_view}(p, id, D, T) \Leftrightarrow \mathrm{Vs\_id}(t_l, p) = \perp) \wedge$
$(\exists i \nexists j \exists id' \exists D' \exists T' : i < j < l \wedge t_i = \textbf{vs\_view}(p, id', D', T') \wedge$
$(t_j = \textbf{recover}(p) \vee \exists id'' \exists D'' \exists T'' : t_j = \textbf{vs\_view}(p, id'', D'', T'')) \Leftrightarrow \mathrm{Vs\_id}(t_l, p) = id \wedge id \neq \perp)$

Proof: From the algorithm, after a **recover** event occurs, Vs_id is immediately set to ⊥. After a **vs_view** event occurs, Vs_id is immediately set to the identifier of that VS view. There are no other cases under which Vs_id is modified. **Vs_view** events only occur in response to **evs_deliver** events. Therefore, by EVS Property 2.1 (Initial View Event) and Lemma 5.2 (VS Views) the identifier for a VS view is the most recently installed EVS view identifier, which by the definition of the **evs_view** event is never ⊥.

6.4 **Lemma (id-vid)** *Evs_id and Vs_id, where defined, are respectively equivalent to evs_vid and vs_vid. Formally:*

$\mathrm{pid}(t_l) = p \wedge t_l \neq \textbf{crash}(p) \wedge t_l \neq \textbf{recover}(p) \Rightarrow \mathrm{evs\_vid}(t_l, p) = \mathrm{Evs\_id}(t_l, p) \wedge$
$\mathrm{vs\_vid}(t_l, p) = \mathrm{Vs\_id}(t_l, p)$

Proof: Lemma 5.1 (Evs_id) proved that Evs_id is ⊥ after a **recover** event and before the first following **evs_view** event. That lemma also proved that Evs_id equals the identifier of the most recent **evs_view** event at the process. Therefore, by the definition of evs_vid, Evs_id, where it is defined, is equivalent to evs_vid. A similar argument is made for Vs_id’s equivalence to vs_vid.

6.5 **Lemma (Vs_Survivors)**

1. *A process’ Vs_Survivors set at a particular event is ∅, if and only if a* **vs_view** *event has not occurred at the process since the most recent* **recover** *event at that process. Formally:*

   $\mathrm{Vs\_Survivors}(t_c, p) = \emptyset \Leftrightarrow \mathrm{pid}(t_c) = p \wedge t_c \neq \textbf{crash}(p) \wedge t_c \neq \textbf{recover}(p) \wedge$
   $\exists a \nexists b \exists id \exists D \exists T : a < b < c \wedge t_a = \textbf{recover}(p) \wedge t_b = \textbf{vs\_view}(p, id, D, T)$

2. *A process’ Vs_Survivors set at a particular event is the intersection of the most recent* **vs_view** *event’s membership set with the transitional sets of the* **evs_view** *events that occurred at this process since that most recent* **vs_view** *event up to the event in question. Formally:*

   $\mathrm{Vs\_Survivors}(t_d, p) = S \wedge p \in S \Leftrightarrow \mathrm{pid}(t_d) = p \wedge t_d \neq \textbf{crash}(p) \wedge t_d \neq \textbf{recover}(p) \wedge$
   $\exists a \nexists b \exists id \exists D \exists T : a < b < d \wedge t_a = \textbf{vs\_view}(p, id, D, T) \wedge$
   $(t_b = \textbf{recover}(p) \vee \exists id' \exists D' \exists T' : t_b = \textbf{vs\_view}(p, id', D', T') \wedge$
   $S = D \cap \forall T'' : \forall c \forall id'' \forall D'' : a < c < d \wedge t_c = \textbf{evs\_view}(p, id'', D'', T'')$

Proof: Immediately after a **recover** event at a process', its Vs_Survivors set is set to the empty set. Thereafter, Vs_Survivors is only modified immediately after **vs_view** and **evs_view** events. In the case of a **vs_view** event, Vs_Survivors is set to the membership set of that VS view. In the case of an **evs_view** event, Vs_Survivors is set to the intersection of itself with the transitional set of the EVS view. This proves that before the first **vs_view** event following a **recover** event at a process, Vs_Survivors is ∅ at that process.

By EVS Property 2.2 (Self-Inclusion) a process is always a member of any EVS view event that it installs. By EVS Property 2.17 (Transitional Set 1,3) a process is always in the transitional set of any EVS view that it installs, except for the first following a **recover** event. Lemma 5.2 (VS Views), EVS Property 2.17 (Transitional Set) and the calculation of the Vs_Survivors set proves that after the first **vs_view** event following the most recent **recover** event at a process, that a process is always in its own Vs_Survivors set. Therefore, after installing the first such VS view, Vs_Survivors is not the empty set, which completes the proof of the first property. By the direct construction of the algorithm, the Vs_Survivors set is equal to the most recent VS view's membership intersected with each subsequent **evs_view** event's transitional set occurring at that process until the next **vs_view** event at the process.

6.6 **Lemma (Flush Messages)** *A process generates at most one flush message marked with a particular view identifier. Formally:*

$$t_b = \textbf{evs\_send}(p, m) \wedge \text{is\_flush\_msg}(m) \wedge \text{flush\_msg\_id}(m) = \text{id} \Rightarrow$$
$$\exists a \exists D \exists T \nexists c \exists m' : a < b \wedge t_a = \textbf{evs\_view}(p, \text{id}, D, T) \wedge$$
$$c \neq b \wedge t_c = \textbf{evs\_send}(p, m') \wedge \text{is\_flush\_msg}(m') \wedge \text{flush\_msg\_id}(m') = \text{id}$$

Proof: Flush messages are only sent in two cases: (1) in response to a request_flush event when Vs_delivd_flush_req is true and Vs_sent_flush is false and (2) in response to an **evs_view** event when Vs_delivd_flush_req is true and Vs_sent_flush is true. Vs_delivd_flush_req and Vs_sent_flush are only set to false after a **vs_view** event occurs at the process.

Immediately after a **recover** event both Vs_delivd_flush_req and Vs_sent_flush are set to true. Therefore, until the next **vs_view** at this process, flush messages are generated only by case (2) and each flush message is marked with the view identifier of the instigating **evs_view** event. Immediately after a **vs_view** event neither case can be triggered. Vs_delivd_flush_req is only[6] set to true in response to the first **evs_view** event following a **vs_view** event at the process. Vs_sent_flush is only [7] set to true in response to a request_flush event when Vs_delivd_flush_req is true and Vs_sent_flush is false, therefore, in this state, only case (1) can generate a flush message. If case (1) is triggered, the flush message is marked with Evs_id, which by Lemma 5.1 (Evs_id) is the view identifier of the most recent EVS view, a **vs_flush** event is generated and Vs_sent_flush is immediately set to true. After that, again, only case (2) can generate flush messages in response to subsequent **evs_view** events until after the next **vs_view** event.

[6] Ignoring Vs_delivd_flush_req's initialization to true upon recovery.
[7] Ignoring Vs_sent_flush's initialization to true upon recovery.

Case (1) only generates one flush message (and a corresponding **vs_flush** event) in a VS view that is marked with the view identifier of the most recent EVS view. Case (2) only generates one flush message per triggering **evs_view** event, marked with the view identifier of that EVS view, in a VS view after case (1) has already been triggered. Therefore, by EVS Property 2.4 (Local Monotonicity) these two cases together produce at most one flush message marked with a particular view identifier.

6.1 **Theorem (VS Initial View Event)** *Every* **vs_flush, vs_flush_req, vs_send, vs_deliver***, and* **vs_trans_sig** *event occurs within some VS view. Formally:*

$t_a = \mathbf{vs\_flush}(p) \vee t_a = \mathbf{vs\_flush\_req}(p) \vee t_a = \mathbf{vs\_send}(p, m) \vee$
$t_a = \mathbf{vs\_deliver}(p, m) \vee t_a = \mathbf{vs\_trans\_sig}(p) \Rightarrow vs\_vid(t_a, p) \neq \perp$

Proof: After a **recover** event Vs_sent_flush, Vs_delivd_flush_req and Vs_delivd_trans_sig are immediately set to true. Vs_sent_flush, Vs_delivd_flush_req, and Vs_delivd_trans_sig are set to false only immediately after a **vs_view** event.

By the algorithm, Vs_sent_flush is false whenever a **vs_flush** event occurs, Vs_delivd_flush_req is false whenever a **vs_flush_req** event occurs, Vs_sent_flush is false whenever a **vs_send** event occurs, and Vs_delivd_trans_sig is false whenever a **vs_trans_sig** event occurs. Therefore, no **vs_flush, vs_flush_req, vs_send** and **vs_trans_sig** events can occur after a **recover** event until after a following **vs_view** event occurs, and therefore due to Lemma 5.2 (VS Views) and Lemma 5.4 (id-vid) the vs_vid of those events is not ⊥.

By the algorithm, messages are **vs_deliver**ed in only two cases: the message is not a flush message and (1) the message is marked with the same identifier as the receiver's Vs_id and the sender is in the receiver's Vs_Survivors set and (2) the message is not marked with the same identifier as the receiver's Vs_id, but it is marked with the receiver's Evs_id and the receiver installs the corresponding VS view.

Due to Lemma 5.2 (VS Views) and Lemma 5.4 (id-vid) the vs_vid of delivery events due to (2) is not ⊥. Above, it was proved that no **vs_send** events occur at a process before the first **vs_view** event following a **recover** event. By Lemma 5.2 (VS Views) this means that no regular message is ever marked with VS view identifier ⊥. Therefore, by Lemma 5.3 (Vs_id) no regular messages can be delivered by a process before it installs its first VS view following a **recover** event because its Vs_id is ⊥ during that time.

6.2 **Theorem (VS Self-Inclusion)** *If a process p installs a view, then p is a member of the membership set. Formally:*

$t_a = \mathbf{vs\_view}(p, id, D, T) \Rightarrow p \in D$

Proof: EVS Property 2.2 (Self-Inclusion) and Lemma 5.2 (VS Views) prove this theorem.

6.3 **Theorem (VS Membership Agreement)** *If a process p installs a view with identifier id and a process q installs a view with the same identifier, then the membership sets of the views are identical. Formally:*

$t_a = \textbf{vs\_view}(p, id, D, T) \wedge t_i = \textbf{vs\_view}(q, id, D', T') \Rightarrow D = D'$

Proof: EVS Property 2.3 (Membership Agreement) and Lemma 5.2 (VS Views) prove this theorem.

6.4 **Theorem (VS Local Monotonicity)** *If a process p installs a view with identifier id' after installing a VS view with identifier id, then id' is greater than id. Formally:*

$t_a = \textbf{vs\_view}(p, id, D, T) \wedge t_b = \textbf{vs\_view}(p, id', D', T') \wedge a < b \Rightarrow id < id'$

Proof: EVS Property 2.4 (Local Monotonicity) and Lemma 5.2 (VS Views) together imply: $t_a = \textbf{vs\_view}(p, id, D, T) \wedge t_b = \textbf{vs\_view}(p, id', D', T') \wedge a < b \Rightarrow id \leq id'$

The algorithm never installs two VS views with the same view identifier. Vs_Flushers is the set of members from which this process has received flush messages marked with the same view identifier as Evs_id. If Vs_Flushers becomes equal to Vs_Members immediately after an **evs_deliver** event, only then is a **vs_view** event is generated. If this process already installed a **vs_view** event with an identifier id, then this process must have received a flush message marked with id from each of the members of id. Lemma 5.6 (Flush Messages) proved that a process sends at most one flush message marked with a particular EVS view identifier. That lemma, together with EVS Property 2.5 (No Duplication) and the fact that Vs_Flushers is cleared after every **vs_view** event proves that a process could not possibly collect the necessary flush messages in order to install a VS view that it had already previously installed.

6.5 **Theorem (VS No Duplication)** *A process never delivers a message more than once. Formally:*

$t_a = \textbf{vs\_deliver}(p, m) \wedge t_b = \textbf{vs\_deliver}(p, m) \Rightarrow a = b$

Proof: As described above, messages are only **vs_deliver**ed in response to **evs_deliver** events. They are either immediately delivered, dropped or they are later popped off of the Delay_Queue and delivered immediately after a **vs_view** event occurs. Messages are only pushed at most once onto the Delay_Queue upon receipt and are at most delivered once – when messages are delivered off of the Delay_Queue they are popped off until the queue is empty. This argument and EVS Property 2.5 (No Duplication) prove the theorem.

6.6 **Theorem (VS Delivery Integrity)** *A* **vs_deliver** *event in a view is the result of a preceding* **vs_send** *event by a member of that view. Formally:*

$t_a = \textbf{vs\_view}(p, id, D, T) \wedge t_b = \textbf{vs\_deliver}(p, m) \wedge \text{vs\_vid}(t_b, p) = id \Rightarrow$
$\exists i \exists q : i < a \wedge t_i = \textbf{vs\_send}(q, m) \wedge q \in D$

Proof: EVS Property 2.6 (Delivery Integrity) ensures that for every **evs_deliver** event there is a preceding **evs_send** event of the same message. The algorithm only sends messages through **evs_send** events without generating an accompanying **vs_send** event when the message being sent is a flush message. However, flush messages are never delivered and the algorithm does not generate any **vs_deliver** events that are not originally caused by **evs_deliver** events, as previously described. This proves that every **vs_deliver** event of a message is preceded by a **vs_send** event of that message. Theorem 5.9 (Sending View Delivery) proves that messages are only **vs_deliver**ed when a message is marked with the same view identifier as the receiver's current VS view. Therefore, if a message is marked in that manner, the sender installed that VS view as well and by Theorem 5.2 (VS Self-Inclusion) must be a member of that VS view.

6.7 Theorem (Flush Requests and Flushes)

1. *At most one* **vs_flush_req** *event occurs in a view at a process. Formally:*

   $t_a = \textbf{vs\_flush\_req}(p) \Rightarrow \nexists b : b \neq a \wedge t_b = \textbf{vs\_flush\_req}(p) \wedge \text{vs\_vid}(t_a, p) = \text{vs\_vid}(t_b, p)$

2. *At most one* **vs_flush** *event occurs in a view at a process. A* **vs_flush** *event is preceded by a* **vs_flush_req** *event in that view at a process. No* **vs_send** *events follow a* **vs_flush** *event in a view at a process. Formally:*

   $t_b = \textbf{vs\_flush}(p) \Rightarrow \nexists d : d \neq b \wedge t_d = \textbf{vs\_flush}(p) \wedge \text{vs\_vid}(t_b, p) = \text{vs\_vid}(t_d, p) \wedge$
   $\exists a \nexists c \exists m : a < b < c \wedge t_a = \textbf{vs\_flush\_req}(p) \wedge t_c = \textbf{vs\_send}(p, m) \wedge$
   $\text{vs\_vid}(t_a, p) = \text{vs\_vid}(t_b, p) = \text{vs\_vid}(t_c, p)$

3. *Every* **vs_view** *event, except for the first following a* **recover** *event, at a process is preceded by a* **vs_flush** *event. Formally:*

   $t_b = \textbf{vs\_view}(p, id, D, T) \Rightarrow$
   $\exists a : a < b \wedge (t_a = \textbf{vs\_flush}(p) \vee t_a = \textbf{recover}(p)) \wedge \text{vs\_vid}(t_a, p) = \text{vs\_vid}(t_b, p)$

Proof: A **vs_flush_req** event only occurs in response to an **evs_view** event when Vs_delivd_flush_req is false. A **vs_flush** event only occurs in response to a request_flush event when Vs_delivd_flush_req is true and Vs_sent_flush is false.

Vs_delivd_flush_req is only set to false immediately after a **vs_view** event and is only[8] set to true immediately after the first **evs_view** event following a **vs_view** event at a process. This proves that only one **vs_flush_req** event occurs per VS view per process. Immediately after a **vs_view** event a **vs_flush** event cannot occur because Vs_delivd_flush_req is set to false. Vs_delivd_flush_req is only set to true when a subsequent **evs_view** event occurs and Vs_delivd_flush_req is false. In this case, immediately before Vs_delivd_flush_req is set to true a **vs_flush_req** event occurs. This proves that any **vs_flush** event at a process is preceded by a **vs_flush_req** event at that process in that VS view.

As stated above, Vs_sent_flush must be false for a **vs_flush** event to occur. Vs_sent_flush is only set to false immediately after a **vs_view** event and is only[9] set to true immediately after a **vs_flush** event occurs. This proves that only one **vs_flush** event

[8] Ignoring Vs_delivd_flush_req's initialization to true upon recovery.

[9] Ignoring Vs_sent_flush's initialization to true upon recovery.

occurs per VS view at a process. When Vs_sent_flush is true, request_send events, which are the only events that can generate **vs_send** events, are illegal and generate an error. This proves that no **vs_send** events can occur after a **vs_flush** event before a following **vs_view** event occurs at that process.

In order to install a VS view a process must collect a flush message marked with the identifier of that view from each of the potential members of that view. Due to Lemma 5.2 (VS Views) and EVS Property 2.2 (Self-Inclusion), this implies that a process itself must send (and receive back) a flush message marked appropriately in order to install a VS view. The proof of Lemma 5.6 (Flush Messages) demonstrated that in order to generate a flush message in a VS view a request_flush event must first occur, which in turn generates a **vs_flush** event. This proof, along with Theorem 5.1 (VS Initial View Event) proves that any **vs_view** event is preceded either by a **recover** or a **vs_flush** event in the same view as the view in which that **vs_view** event is delivered.

6.8 **Theorem (VS Self-Delivery)** *If a process p sends a message m, then p delivers m unless it crashes. Formally:*

$t_a = \textbf{vs\_send}(p, m) \land \nexists b : a < b \land t_b = \textbf{crash}(p) \Rightarrow \exists c : t_c = \textbf{vs\_deliver}(p, m)$

Proof: A process will immediately **vs_deliver** its own non-flush message received in an **evs_deliver** event of that message. Messages are only immediately delivered if the message is marked with the identifier of the receiver's current VS view and the sender is in the receiver's Vs_Survivors set.

Theorem 5.1 (VS Initial View Event) proved that no **vs_send** events can occur at a process until after the first **vs_view** event following a **recover** event at that process. In the proof of Theorem 5.7 (Flush Requests and Flushes) it was shown that a **vs_flush** event must occur at this process before it can install any following **vs_view** events. Furthermore, it was also shown that no **vs_send** events occur in a VS view after a **vs_flush** event and before the following **vs_view** event. The flush messages generated by the **vs_flush** event and possibly subsequent **evs_deliver** events are FIFO messages. Therefore, by EVS Property 2.7 (Self-Delivery), EVS Property 2.12 (FIFO Messages) and EVS Property 2.9 (Sane View Delivery) before a flush message generated in this VS view by this process could be delivered back to this process, all previous messages sent in that VS view by this process must be received by it. Therefore, because any message sent by the process since its last **vs_view** event is marked with that view's identifier and because the process can not install another VS view until it receives back its own flush message, all messages sent by that process in that VS view will be **evs_deliver**ed in the process' current VS view (i.e. - will match its Vs_id). Lemma 5.5 (Vs_Survivors) proved that a process is always in its own Vs_Survivors set after installing the first VS view following a **recover** event. Therefore, a process will always immediately deliver its own non-flush messages upon receipt. This argument and EVS Property 2.7 (Self-Delivery) prove this theorem.

6.9 **Theorem (Sending View Delivery)** *Messages are delivered in the view in which they are sent. Formally:*

$t_a = \textbf{vs\_deliver}(p, m) \wedge vs\_vid(t_a, p) = id \Rightarrow \exists i \exists q : t_i = \textbf{vs\_send}(q, m) \wedge vs\_vid(t_i, q) = id$

Proof: By the algorithm, messages are **vs_deliver**ed in only two cases: the message is not a flush message and (1) the message is marked with the same identifier as the receiver's Vs_id and the sender is in the receiver's Vs_Survivors set and (2) the message is not marked with the same identifier as the receiver's Vs_id, but it is marked with the receiver's Evs_id and the receiver installs the corresponding VS view. Messages that meet criteria (2) are placed in Delay_Queue and are only delivered after installing the corresponding VS view.

From the algorithm, if a message is **vs_deliver**ed by matching criteria (1) then it was sent in the same VS view because messages are marked with the identifier of the VS view in which they are sent. If a message is **vs_deliver**ed due to criteria (2) then it was sent in the VS view that this process installs immediately before delivering it. This is because the message was marked with the same identifier as the receiver's Evs_id. If the receiver installs a VS view without any intervening **evs_view** events occurring then its new VS view identifier is Evs_id by Lemma 5.2 (VS Views). The messages in the Delay_Queue are then delivered in the VS view in which they were sent. If an intervening **evs_view** event had occurred, then the algorithm would drop all of the messages in Delay_Queue. By Theorem 5.2 (VS Self-Inclusion), this proves that when any message is **vs_deliver**ed, the recipient's Vs_id matches the identifier on the message and that the sender is in the recipient's Vs_Survivors set.

6.10 **Theorem (VS Transitional Set)**

1. *The transitional set for the first view installed at a process following a* **recover** *event is the empty set. Formally:*

   $t_a = \textbf{vs\_view}(p, id, D, T) \wedge vs\_vid(t_a, p) = \perp \Rightarrow T = \varnothing$

Proof: The transitional set of a **vs_view** event at a process is the Vs_Survivors set at that process when the last necessary flush message to install that VS view is received. Lemma 5.5. (Vs_Survivors) proved that Vs_Survivors is the empty set only before the first **vs_view** event at a process following the most recent **recover** event at that process.

2. *If a process p installs a view in a previous view, then the transitional set for the new view at p is the union of p with a subset of the intersection between the two views' membership sets. Formally:*

   $t_a = \textbf{vs\_view}(p, id, D, T) \wedge t_b = \textbf{vs\_view}(p, id', D', T') \wedge vs\_vid(t_b, p) = id \Rightarrow$
   $p \in T' \wedge T' \subseteq D \cap D'$

Proof: From the algorithm, the transitional set of a **vs_view** at a process is the Vs_Survivors set at that process when the last necessary flush message is **evs_deliver**ed. Lemma 5.5 (Vs_Survivors) proved that a process is always a member of its own Vs_Survivors set after installing the first VS view following the most recent **recover** event. That lemma also proved that at any point, Vs_Survivors is the intersection of the most recent VS view's membership with the transitional sets of the subsequent EVS views up until that point (assuming no intervening crashes). Therefore, by Lemma 5.2 (VS Views) and EVS Property 2.17 (Transitional Set), the transitional set of EVS view id' is a subset of the membership of VS view id'. Furthermore, Vs_Survivors is set to the membership set of the most recent previous VS view immediately after that view is installed. Therefore by Lemma 5.5 (Vs_Survivors) and EVS Property 2.17 (Transitional Set), the transitional set of a VS view at a process is a subset of the intersection between that VS view's membership set and the previous VS view's membership sets and always contains this process.

3. *If processes p and q install the same view and q is included in p's transitional set for that view, then p's previous view was identical to q's previous view. Formally:*

   $t_a = \textbf{vs\_view}(p, id', D, T) \wedge vs\_vid(t_a, p) = id \wedge t_i = \textbf{vs\_view}(q, id', D', T') \wedge q \in T \Rightarrow vs\_vid(t_i, q) = id$

Proof: If q is in p's transitional set for a VS view, then by Lemma 5.2 (VS Views), Lemma 5.5 (Vs_Survivors) this implies that q was in the transitional set of every EVS view at p that followed id up to and including EVS view id'. From the assumption, q installed VS view id', which by Lemma 5.2 (VS Views) implies it also installed EVS view id'. From EVS Property 2.17 (Transitional Set), because p and q install the same EVS view id' and q is in p's transitional set, then q installed the same previous EVS view and p was in q's transitional set for id'. Now a reverse iterative argument can be made for the chain of EVS views that occurred between the two VS views at p – because both processes eventually install the same EVS view and one of the processes has the other in its EVS transitional set for that view, then they both installed the same previous EVS view. This argument can be repeated back along the chain of EVS views starting with EVS view id' all the way back to EVS view id.

I have shown that both processes installed EVS view id and all subsequent EVS views up to and including id', and from EVS Property 2.17 (Transitional Set 3) that they were in each other's EVS transitional sets throughout that chain of views. From the definition of EVS vsynchronous_in, both p and q were vsynchronous_in in EVS view id, which by EVS Property 2.10 (Virtual Synchrony) implies that they delivered the same set of messages in that view. Delivering a **vs_view** event is only dependent on receiving a flush message from each of the potential members of the next VS view in the corresponding EVS view. Therefore, due to the algorithm, EVS Property 2.4 (Local Monotonicity), EVS Property 2.9 (Same View Delivery) and EVS Property 2.10 (Virtual Synchrony) process q also received all of the flush messages in EVS view id and therefore installed VS view id. Also, because q was virtually synchronous to p throughout the chain of EVS views that they moved through together, q could not install any VS views that p did not. Therefore, q installed VS view id' in VS view id.

4. *If processes p and q install the same view and q is included in p's transitional set for that view, then p and q have the same transitional sets for that view. Formally:*

   $t_a = \textbf{vs\_view}(p, id', D, T) \wedge \text{vs\_vid}(t_i, p) = id \wedge t_i = \textbf{vs\_view}(q, id', D, T') \wedge q \in T \Rightarrow T = T'$

Proof: Theorem 5.10 (VS Transitional Set) proved that if two processes install the same VS view and one of the processes is in the others transitional set for that view, then they both installed the same previous VS view. Furthermore, the theorem proved that they were EVS virtually synchronous throughout the chain of EVS views that they moved through together before installing their next VS view. EVS Property 2.17 (Transitional Set 4) states that if two processes install the same next EVS view in the previous view, then they have the same transitional sets for the new view. The transitional set of a VS view is Vs_Survivors at the point of installation. Vs_Survivors is immediately set to the membership set of a VS view upon installation and is then intersected with any subsequent EVS views' transitional sets until the next VS view at the process. Because both processes install the same previous VS view id and move together through the same chain of EVS views before they both install their next VS view id', they see the same transitional set for each EVS view, by EVS Property 2.17 (Transitional Set 4). Therefore, due to EVS (Membership Agreement), EVS Property 2.17 (Transitional Set) and Lemma 5.5 (Vs_Survivors), throughout the chain of intermediate EVS views and at the point of installing VS view id', their Vs_Survivors sets are virtually synchronously identical and, therefore, their transitional sets for the VS view id' are identical.

6.11 **Theorem (VS Sane View Delivery)**

1. *A message is not delivered in a view earlier than the one in which it was sent. Formally:*

   $t_a = \textbf{vs\_send}(p, m) \wedge \text{vs\_vid}(t_a\ p) = id \wedge t_i = \textbf{vs\_deliver}(q, m) \wedge \text{vs\_vid}(t_i, q) = id' \Rightarrow id \leq id'$

Proof: Theorem 5.9 (VS Sending View Delivery) proves this theorem.

2. *If a process p sends a message m, crashes and later recovers in a view id and a process q delivers m, then m is delivered in a view before id. Formally:*

   $t_a = \textbf{vs\_send}(p, m) \wedge t_b = \textbf{crash}(p) \wedge t_c = \textbf{vs\_view}(p, id, D, T) \wedge a < b < c \wedge$
   $\text{vs\_vid}(t_c, p) = \perp \wedge t_i = \textbf{vs\_deliver}(q, m) \Rightarrow \text{vs\_vid}(t_i, q) < id$

Proof: Theorem 5.9 (VS Sending View Delivery) proved that messages are only delivered in the view in which they are sent. Message m is sent by p in a VS view installed at p earlier than id, therefore, Theorem 5.4 (VS Local Monotonicity) proves this theorem.

3. *If two messages m and m' are sent, respectively, by processes p and p' such that the send of m' is causally preceded by the send of m and a process q' delivers both messages, then q' does not deliver m in a later view than m'. Formally:*

   $t_a = \textbf{vs\_send}(p, m) \wedge t_d = \textbf{vs\_send}(p', m') \wedge t_a \rightarrow t_d \wedge$
   $t_j = \textbf{vs\_deliver}(q', m) \wedge t_k = \textbf{vs\_deliver}(q', m') \Rightarrow \text{vs\_vid}(t_j, q') \leq \text{vs\_vid}(t_k, q')$

Proof: In order for m to be sent causally before m', it must be sent in a VS view with identifier less than or equal to the sending view of m' by Theorem 5.4 (VS Local Monotonicity). Theorem 5.9 (Sending View Delivery), therefore, proves this theorem.

6.7 **Lemma (VS Message Ordering)** *All messages delivered by the VS algorithm maintain their respective ordering properties.*

Proof: If it can be shown that the reordering of messages that the algorithm performs does not violate any of the ordering guarantees of the EVS properties, then because **vs_send** events are, effectively, **evs_send** events and **vs_deliver** events of message only occur as a result of earlier **evs_deliver** events of those messages, then the algorithm implicitly maintains the EVS ordering guarantees on the messages it delivers. The only reordering of messages delivered by **evs_deliver** events occurs when messages are pushed onto the Delay_Queue to potentially be **vs_deliver**ed in a later VS view. For all other messages, the algorithm either delivers them immediately when they are **evs_deliver**ed or immediately drops and never delivers them.

The messages that are pushed onto Delay_Queue can only be non-causal messages. From the algorithm, messages are marked with the VS view identifier in which they were sent. For a message to be pushed onto Delay_Queue, the received message must be marked with the receiver's current Evs_id, which is different than its current Vs_id. By the algorithm, this implies that the sender already installed a VS view with the same identifier as the receiver's Evs_id and then sent a message.

From the algorithm, in order to install that VS view the sender must have received an appropriately marked flush message from each of the potential members of the VS view. Because the message's view identifier matches the receiver's Evs_id, by Lemma 5.2 (Evs_id) and EVS Property 2.2 (Self-Inclusion) the receiver of the causal message is one of the view's potential members. If the sender sent a message in the new VS view, then it must have sent the message after receiving all of the flush messages, including the one from the receiver. By EVS Property 2.13 (Causal Messages) only two cases are possible for a causal message sent after receiving all of those flush messages: (1) before the recipient could receive the causal message it would receive all of the flush messages and, by the algorithm, install the same VS view or (2) the receiver received one or more **evs_view** events before receiving all of the flush messages for the VS view in question. Case (1), conflicts with the Delay_Queue push assumption because the Causal message's view identifier would either match the receiver's Vs_id or, by EVS Property 2.4 (Local Monotonicity) and Lemma 5.2 (Evs_id), it would not match the receiver's Evs_id. In case (2), by the algorithm, the receiver does not install that VS view. Furthermore, by EVS Property 2.4 (Local Monotonicity) and Lemma 5.2 (VS Views) its Evs_id is now higher than the sender's installed view and its Vs_id will never match the sender's. Therefore, the receiver drops the causal message. In neither of the two possible cases will a causal message ever be placed in Delay_ Queue. Therefore, only non -causal messages can be placed in Delay_Queue.

The only way the EVS ordering properties could be violated is if by postponing the **vs_deliver**y of messages in Delay_Queue the algorithm violated the FIFO, Causal or Agreed ordering properties. The above argument showed that the postponing of these non-causal messages can not violate the Causal or Agreed VS ordering properties. This is because any messages that are causally dependent on messages in Delay_Queue would also be causally dependent on the flush messages for that VS view. Therefore, these causal messages would only be received after this process either installs that VS view and **vs_deliver**s all of the messages in Delay_Queue or it drops the messages in Delay_Queue. The only way the postponing could violate the FIFO ordering property is if a FIFO message was delivered from a sender, when one of the messages in Delay_Queue was sent by that sending process <u>before</u> the FIFO message.

Assume that this ordering violation occurs. This implies that the later FIFO message was marked with the receiver's current Vs_id (which is different than its Evs_id), that the sender was in the recipient's Vs_Survivors set and that the previously sent message was marked with the recipient's Evs_id. But, by Lemma 5.2 (Vs Views) and EVS Property 2.4 (Local Monotonicity) the recipient's Evs_id is greater than the recipient's Vs_id. Messages are marked with the identifier of the VS view in which they are sent. Therefore, by Theorem 5.4 (VS Local Monotonicity) and EVS Property 2.12 (FIFO Messages) this violation could not happen.

Messages in Delay_Queue are delivered upon installing a VS view and the queue is cleared after every **evs_view** event. Messages are pushed onto the end of the queue as they are received and popped off of the front as they are **vs_deliver**ed, therefore, any FIFO ordering of the messages in the queue is maintained. Since the only reordering of messages delivered by **evs_deliver** events is compatible with all of the message types' ordering guarantees and all other messages are delivered in the order in which they are received or dropped, the ordering guarantees provided by the respective EVS Properties are maintained.

6.12 **Theorem (VS Virtual Synchrony)** *If processes p and q are virtually synchronous in a view, then any message delivered by p in that view is also delivered by q. Formally:*

$\text{vs\_vsynchronous\_in}(p, q, id) \wedge t_a = \textbf{vs\_deliver}(p, m) \wedge \text{vs\_vid}(t_a, p) = id \Rightarrow$
$\exists i : t_i = \textbf{vs\_deliver}(q, m)$

Proof: Theorem 5.10 (VS Transitional Set) proved that if two processes install the same VS view and one of the processes is in the other's transitional set for that view, then they both installed the same previous VS view. Furthermore, the theorem proved that they were virtually synchronous throughout the chain of EVS views that they moved through together before installing their next VS view. From EVS Property 2.10 (Virtual

Synchrony), these two processes received the same set of messages in each EVS view starting with the EVS view corresponding to the first VS view installed (id) up to the last EVS view corresponding to the second VS view they installed (id').

Any messages sent in VS view id that were received in EVS view id would be delivered by both p and q. By EVS Property 2.9 (Sane View Delivery) and Lemma 5.2 (VS Views), the EVS views in which messages sent in VS view id are delivered are greater than or equal to id. Any messages received in EVS view id would be received by p or q either before it installed VS view id or after it installed VS view id. If such a message was received in EVS view id before the process installed VS view id, then by the assumption and Lemma 5.2 (VS Views) the identifier marked on the message would match the process' Evs_id and be different than its Vs_id. In this case, the algorithm buffers the message in Delay_Queue and delivers it after it installs VS view id, which by the assumption it does. If any message was received in EVS view id after the process installed VS view id, then the message would be marked with the receiver's Vs_id and the sender would be in the receiver's Vs_Survivors set, by Lemma 5.5 (Vs_Survivors), Theorem 5.2 (VS Self-Inclusion) Theorem 5.3 (VS Membership Agreement) and, therefore, the message would be delivered.

Theorem 5.10 (VS Transitional Set) proved that the two processes had virtually synchronously identical Vs_Survivors sets and Vs_ids in VS view id throughout the chain of EVS views [id, id'] after they installed VS view id. Therefore, because any messages sent in VS view id that are received in EVS view id are delivered and in later EVS views p and q have virtually synchronously the same Vs_ids and Vs_Survivors sets, any messages that p or q delivered in EVS views in the view range [id, id'), they both delivered.

From the assumption, both p and q installed VS view id'. In order for this to happen, both processes must have collected appropriate flush messages from each of the members of id' in EVS view id'. Since flush messages are at least FIFO messages, then by EVS Property 2.12 (FIFO Messages) any messages sent by these processes prior to their flush messages that were delivered in EVS view id' must have already been delivered to both p and q before they installed VS view id'. Therefore, due to the fact that regular messages for a VS view cannot be sent by a process after it sends a flush message in that view and any messages not destined for VS view id would be either dropped or buffered, both p and q receive the same set of messages sent in VS view id in EVS view id' before installing VS view id'. Since p and q still have the same Vs_id and Vs_Survivors sets while receiving those messages, as shown above, they both deliver the same set of those messages in the previous VS view.

6.13 **Theorem (VS Transitional Signals)**

1. *At most one* **vs_trans_sig** *event occurs at a process per view. Formally:*

   $t_a = \mathbf{vs\_trans\_sig}(p) \wedge vs\_vid(t_a, p) = id \Rightarrow$
   $\nexists b : b \neq a \wedge t_b = \mathbf{vs\_trans\_sig}(p) \wedge vs\_vid(t_b, p) = id$

Proof: **Vs_trans_sig** events are only generated when Vs_delivd_trans_sig is false. This variable is only set to false after every **vs_view** event and is immediately set to true whenever a **vs_trans_sig** event occurs. Therefore, at most one **vs_trans_sig** event can be generated per VS view.

2. *If two processes p and q are virtually synchronous in a VS view, id, and p has a* **vs_trans_sig** *event occur in that view, then q also has a* **vs_trans_sig** *event occur in that view and they both deliver the same sets of agreed messages before and after their* **vs_trans_sig** *events. Formally:*

$$\begin{aligned}
&\text{vs\_vsynchronous\_in}(p, q, id) \wedge t_b = \mathbf{vs\_trans\_sig}(p) \wedge \text{vs\_vid}(t_b, p) = id \Rightarrow \\
&\exists j : t_j = \mathbf{vs\_trans\_sig}(q) \wedge \text{vs\_vid}(t_j, q) = id \wedge \\
&(\exists a \exists m : a < b \wedge t_a = \mathbf{vs\_deliver}(p, m) \wedge \text{vs\_vid}(t_a, p) = id \wedge \text{agreed}(m) \Leftrightarrow \\
&\ \exists i \exists m : i < j \wedge t_i = \mathbf{vs\_deliver}(q, m) \wedge \text{vs\_vid}(t_i, q) = id \wedge \text{agreed}(m)) \\
&(\exists c \exists m' : b < c \wedge t_c = \mathbf{vs\_deliver}(p, m') \wedge \text{vs\_vid}(t_c, p) = id \wedge \text{agreed}(m') \Leftrightarrow \\
&\ \exists k \exists m' : j < k \wedge t_k = \mathbf{vs\_deliver}(q, m') \wedge \text{vs\_vid}(t_k, q) = id \wedge \text{agreed}(m'))
\end{aligned}$$

Proof: If two processes are virtually synchronous in a VS view, then by Theorem 5.12 (VS Virtual Synchrony) they both deliver the same set of messages in that view and are virtually synchronous through the same chain, if any, of intermediate EVS views, before installing the following VS view. From the algorithm, **vs_trans_sig** events are only generated in three cases: (1) an **evs_view** event removes a member from the process' Vs_Survivors set, or an **evs_trans_sig** occurred in the current EVS view and no **vs_trans_sig** event has yet occurred in the current VS view when (2) an agreed message is subsequently **vs_deliver**ed or (3) a subsequent **evs_view** event occurs.

It has been shown that two processes that are virtually synchronous in a VS view have virtually synchronously identical Vs_Survivors sets throughout that VS view. Therefore, if an **evs_view** event caused one process to deliver a **vs_trans_sig**, the other process also delivers a **vs_trans_sig** event. Furthermore, because they have delivered the same set of Agreed messages in that VS view up to that point, by EVS Property 2.10 (Virtual Synchrony) and EVS Property 2.9 (Same View Delivery), they deliver the same set of Agreed messages before and after the **vs_trans_sig** event.

From EVS Property 2.16 (Transitional Signals), since the two processes were virtually synchronous through the chain of EVS views, they both receive the same transitional signals in those EVS views (if any) with the same set of agreed messages in each view before and after each signal. Therefore, if a transitional signal was generated in one of those EVS views, both processes would receive it. If they **vs_deliver**ed any subsequent agreed messages in that VS view, then by Theorem 5.12 (VS Virtual Synchrony) and the algorithm they would both deliver a VS transitional signal immediately before the same message in that VS view. If there was an **evs_view** event before they installed their following VS view, then by EVS Property 2.10 (Virtual Synchrony) they would both generate a VS transitional signal at the same point in the their stream of VS agreed messages. If neither of these cases occurred then by the algorithm neither process would generate a transitional signal in the VS view in which they were virtually synchronous.

6.14 **Theorem (VS Reliable Messages)** *All messages are reliable. The Self-Delivery, Same View Delivery and Virtual Synchrony properties implicitly define the safety properties of reliable messages. Formally:*

$$\text{reliable}(m) \equiv m.\text{type} \in \{ R, F, C, A, S \}$$

Proof: The algorithm does not affect messages' types. Theorem 5.8 (VS Self-Delivery), Theorem 5.9 (VS Sending View Delivery) and Theorem 5.12 (VS Virtual Synchrony) satisfy the definition of VS Reliable Messages.

6.15 **Theorem (VS FIFO Messages)**

1. *FIFO messages are reliable messages. Formally:*

$$\text{fifo}(m) \equiv m.\text{type} \in \{ F, C, A, S \}$$

Proof: The algorithm does not affect messages' types.

2. *If a process sends a FIFO message after a previous message, then these messages are delivered in the order in which they were sent at every process that delivers both. Formally:*

$$t_a = \mathbf{vs\_send}(p, m) \wedge t_b = \mathbf{vs\_send}(p, m') \wedge a < b \wedge \text{fifo}(m') \wedge$$
$$t_i = \mathbf{vs\_deliver}(q, m) \wedge t_j = \mathbf{vs\_deliver}(q, m') \Rightarrow i < j$$

Proof: Lemma 5.7 (VS Message Ordering) proves this theorem.

3. *If a process p sends a FIFO message m' after a previous message m and a process q' delivers m', then if any process delivers m, then q' either delivers m or installs a view without p in its transitional set between the delivery views of m and m', or if no process delivers m, then p crashed between sending m and m' and q' installs a view without p in its transitional set between the recovery view of p and the delivery view of m'. Formally:*

$$t_a = \mathbf{vs\_send}(p, m) \wedge t_c = \mathbf{vs\_send}(p, m') \wedge a < c \wedge \text{fifo}(m') \wedge t_l = \mathbf{vs\_deliver}(q', m') \Rightarrow$$
$$(\exists i \exists q : t_i = \mathbf{vs\_deliver}(q, m) \Rightarrow (\exists j : t_j = \mathbf{vs\_deliver}(q', m)) \vee$$
$$(\exists k \exists id' \exists D' \exists T' : t_k = \mathbf{vs\_view}(q', id', D', T') \wedge p \notin T' \wedge \text{vs\_vid}(t_i, q) < id' \leq \text{vs\_vid}(t_l, q'))) \wedge$$
$$(\nexists i \exists q : t_i = \mathbf{vs\_deliver}(q, m) \Rightarrow \exists b \exists id \exists D \exists T : a < b < c \wedge t_b = \mathbf{vs\_view}(p, id, D, T) \wedge \text{vs\_vid}(t_b, p) = \bot \wedge$$
$$\exists k \exists id' \exists D' \exists T' : t_k = \mathbf{vs\_view}(q', id', D', T') \wedge p \notin T' \wedge id \leq id' \leq \text{vs\_vid}(t_l, q'))$$

Proof: Assume m and m' were sent in the same VS view id. In this case q' delivers m. Assume that this is a false statement. Lemma 5.7 (VS Message Ordering) proved that any messages sent in VS view id that are delivered in EVS view id are delivered by any member process that receives them. Therefore, by the assumption m is not delivered by the EVS system in EVS view id, which implies that q' already installed VS view id if it received m. The fact that q' delivers m' implies that p was a member of its Vs_Survivors set when it received m'. Lemma 5.5 (Vs_Survivors), therefore, implies that there was not an EVS view installed at q' in the range (id, EVS delivery view of m'] that did not have p in its transitional set. Therefore, EVS Property 5.12 (FIFO Messages 3) implies that q' received m. Furthermore, EVS Property 5.12 (FIFO Messages 2) implies m was received by q’ before m’. Therefore, by Lemma 5.5 (Vs_Survivors) p was in q’’s Vs_Survivors set when it received and consequently delivered m, which contradicts the assumption.

Assume that m and m’ were sent in different VS views id and id’. Assume that some process delivers m, but q’ does not. The only way the axiom can hold then is if q’ installs a VS view without p in its transitional set between id and id’. Assume that this event does not occur. Then for every VS view that q’ installs in the open range of VS views (id, id’] p’ must be is in its transitional set. From Theorem 5.9 (Sending View Delivery), both p and q’ install VS view id’. Therefore, p and q’ install the same chain of VS views in the range [id, id’] and by Theorem 5.12 (VS Virtual Synchrony) they deliver the same set of messages delivered in those views in the open range [id, id’). Theorem 5.8 (VS Self-Delivery) proved that a process must deliver the messages it sends in a VS view before installing any subsequent VS views. Therefore, both p and q’ deliver m in id, which contradicts the assumption.

Assume that m and m’ were sent in different VS views id and id’. Assume that no process delivers m. In this case the only way the axiom can hold is if q’ installs a VS view without q’ in its transitional set between the recovery view id* of p and the delivery view of m’. Assume that this event does not occur. This implies that all of the VS views installed at q’ with identifiers in the range [id*, id’] had p in the transitional set. Since both processes install VS view id’, p’ and q’ install the same set of VS views in the range [id*, id’]. However, p’ cannot be in q’’s transitional set for VS view id*. This is because p’ is recovering from a crash and therefore has an empty transitional set. Therefore, Theorem 5.10 (VS Transitional Set 1,3) forces q’ not to have p’ in its transitional set, which contradicts the assumption.

6.16 **Theorem (VS Causal Messages)**

1. *Causal messages are FIFO messages. Formally:*

   $\text{causal}(m) \equiv m.\text{type} \in \{ C, A, S \}$

Proof: The algorithm does not affect messages’ types.

2. *If a process sends a causal message m’ such that the send of another message m causally precedes the send of m’, then any process that delivers both messages delivers m before m’. Formally:*

   $t_a = \textbf{vs\_send}(p, m) \wedge t_d = \textbf{vs\_send}(p’, m’) \wedge t_a \rightarrow t_d \wedge \text{causal}(m’) \wedge$
   $t_i = \textbf{vs\_deliver}(q, m) \wedge t_j = \textbf{vs\_deliver}(q, m’) \Rightarrow i < j$

Proof: Lemma 5.7 (VS Message Ordering) proves this theorem.

3. *If a process p’ sends a Causal message m’ such that the send of another message m causally precedes the send of m’, then if any process delivers m, then q’ either delivers m or installs a view without p’ in its transitional set between the delivery views of m and m’, or if no process delivers m, then p crashed between sending m*

and m' and q' installs a view without p in its transitional set between the recovery view of p and the delivery view of m'. Formally:

$$\begin{array}{l}
t_a = \mathbf{vs\_send}(p, m) \wedge t_d = \mathbf{vs\_send}(p', m') \wedge t_a \rightarrow t_d \wedge \text{causal}(m') \wedge t_l = \mathbf{vs\_deliver}(q', m') \Rightarrow \\
(\exists i \exists q : t_i = \mathbf{vs\_deliver}(q, m) \Rightarrow (\exists j : t_j = \mathbf{vs\_deliver}(q', m)) \vee \\
\ (\exists k \exists id' \exists D' \exists T' : t_k = \mathbf{vs\_view}(q', id', D', T') \wedge p' \notin T' \wedge \text{vs\_vid}(t_i, q) < id' \leq \text{vs\_vid}(t_l, q'))) \wedge \\
(\nexists i \exists q : t_i = \mathbf{vs\_deliver}(q, m) \Rightarrow \exists b \exists id \exists D \exists T : a < b < c \wedge t_b = \mathbf{vs\_view}(p, id, D, T) \wedge \text{vs\_vid}(t_b, p) = \perp \wedge \\
\ \exists k \exists id' \exists D' \exists T' : t_k = \mathbf{vs\_view}(q', id', D', T') \wedge p' \notin T' \wedge id \leq id' \leq \text{vs\_vid}(t_l, q'))
\end{array}$$

Proof: Assume m and m' were sent in the same VS view id. In this scenario, q' delivers m. The fact that q' delivers m' implies that p' was in its Vs_Survivors set when it received m'. Lemma 5.5 (Vs_Survivors), therefore, implies that there was not an EVS view installed at q' in the range (id, EVS delivery view of m'] that did not have p in its transitional set. By EVS Property 2.13 (Causal Messages) this implies q' received all of the messages sent in VS view id whose sends causally preceded m', including m. Assume q' does not deliver m. Lemma 5.7 (VS Message Ordering) proved that any messages sent in VS view id that are delivered in EVS view id are delivered by any member process that receives them. Therefore, by the assumption m is not delivered by the EVS system in EVS view id, which implies that q' already installed VS view id before it received m.

The fact that p' was in q''s Vs_Survivors set when it received m', implies, by EVS Property 2.9 (Sane View Delivery) and Lemma 5.5 (Vs_Survivors) that p' was in its Vs_Survivors set when it received m. Therefore if p' sent m, q' would deliver m upon receipt, which violates the assumption. If p' VS delivered m then p' and q' would receive m in the same EVS view, by EVS Property 2.8 (Same View Delivery) and because p' is in q''s Vs_Survivors set when it receives m, they were virtually synchronous in the EVS views [id, delivery view of m). Therefore, by Lemma 5.5 (Vs_Survivors) and EVS Property 2.17 (Transitional Set) they would have the same Vs_Survivors sets when they receive m. Therefore, because p' VS delivers m, so would q', which violates the assumption. The only other way m' could be causally preceded by m is if p' VS delivered a message m*, sent by process p*, whose send causally preceded the send of m' and was causally preceded by the send of m.

The fact that p' delivered m* implies that p* was a member of its Vs_Survivors set when it received m*. Processes p' and q' received m* in the same view, by EVS Property 2.8 (Same View Delivery). Q' received m' in a view no earlier than the delivery view of m*, by EVS Property 2.9 (Sane View Delivery 3), therefore, p' was in q''s Vs_Survivors set when it received m*. Because both p' and q' installed the EVS delivery view of m* and because p' was in q''s Vs_Survivors set when it received m*, p' and q' were virtually synchronous in the sequence of EVS views [id, delivery view of m*). Therefore, by Lemma 5.2 (Vs_Survivors) and EVS Property 2.17 (Transitional Set), p' and q' had the same Vs_Survivors sets when they received m* and, therefore, q' also VS delivered m*, which implies p* is in q''s Vs_Survivors set when it receives m*.

This implies, by EVS Property 2.9 (Sane View Delivery), that if p* sent m, then q' would deliver m upon receipt, which violates the no delivery assumption. If p* VS delivered m, then p* and q' would receive m in the same EVS view, by EVS Property 2.8 (Same View Delivery). Furthermore, because p* is in q''s Vs_Survivors set when it receives m, again by EVS Property 2.9 (Sane View Delivery), they were virtually synchronous in the EVS views [id, delivery view of m). Therefore, by Lemma 5.5 (Vs_Survivors) and EVS Property 2.17 (Transitional Set) they would have the same Vs_Survivors sets when they receive m. Therefore because p^* VS delivers m, so would q', which contradicts the assumption. The only other way m' could be causally preceded by m is if p' VS delivered a message m^{**}, sent by process p^{**}, whose send causally preceded the send of m^* and was causally preceded by the send of m.

This argument can be iteratively applied back along the chain of causally preceding messages between m and m'. At every step, the assumption forces the sender of a message in this causal chain to have not sent and not VS delivered m. Therefore , because there are a finite number of messages in the causal chain of precedence, a contradiction is eventually reached where the send of m cannot causally precede the send of m'. Therefore, if m and m' are sent in the same view such that the send of m causally precedes the send of m' and a process delivers m', it delivers all messages sent in that view that causally precede m'.

Assume that m and m' were sent in different VS views id and id'. Assume that a process delivers m and that q' does not deliver m. In this case the only way the axiom can hold is if q' installs a VS view without p' in its transitional set between the delivery views of m and m'. Assume that this event does not occur. This implies that all of the VS views installed at q' with identifiers in the open range (id, id'] had p' in the transitional set. Both processes p' and q' install id', which implies that the two processes go through the same set of VS views in the range [id, id'] and that they are virtually synchronous in the VS views that they installed in the open range [id, id'). Therefore, q' delivers m if and only if p' delivers m. By Lemma 5.5 (Vs_Survivors), because p' and q' are in each others transitional sets for each of the VS views they installed in the range (id, id'] they were in each others transitional sets for of the EVS views they installed in the range (id, id']. Furthermore, the fact that p' was in q''s Vs_Survivors set when it received m' in EVS view id'', implies that there was not an EVS view installed at q' in the range (id, id''] that did not have p' in the transitional set. By EVS Property 2.13 (Causal Messages) this implies that q' received all of the messages whose sends Causally preceded the send of m' that were EVS delivered in the EVS views that they installed in the range (id, id''], including m. Now the exact same iterative argument used above to prove that if m and m' were sent in the same VS view then q' delivered m, can be applied to this situation, with the additional complexity of multiple Vs_Survivors sets for different VS views.

Assume that m and m' were sent in different VS views id and id'. Assume that no process delivers m. In this case the only way the axiom can hold is if q' installs a VS view without q' in its transitional set between the recovery view of p' and the delivery view of m'. Assume that this event does not occur. This implies that all of the VS views installed at q' with identifiers in the range [id, id'] had p' in the transitional set. Since both processes install VS view id' and p' is in q''s transitional set for that view, they installed the same set of VS views in the range [id, id']. However, p' cannot be in q''s transitional set for VS view id. This is because p' is recovering from a crash and therefore has an empty transitional set. Therefore, Theorem 5.10 (VS Transitional Set 1,3) forces q’ not to have p’ in its transitional set, which contradicts the assumption.

6.17 **Theorem (VS Agreed Messages)**

1. Agreed messages are causal messages. Formally:

   $\mathrm{agreed}(m) \equiv m.\mathrm{type} \in \{ A, S \}$

Proof: The algorithm does not affect messages’ types.

2. *If a process p delivers an agreed message m’, then after that event it will never deliver a message that has a lower ord value. Formally:*

   $t_a = \mathbf{vs\_deliver}(p, m) \wedge t_b = \mathbf{vs\_deliver}(p, m') \wedge \mathrm{agreed}(m) \wedge \mathrm{ord}(m) < \mathrm{ord}(m') \Rightarrow a < b$

Proof: Lemma 5.7 (VS Message Ordering) proves this theorem.

3. *If a process p delivers an agreed message m’ before a* **vs_trans_sig** *event in its current view, then p delivers every message with a lower ord value than m’ delivered in that view by any process. Formally:*

   $t_c = \mathbf{vs\_deliver}(p, m') \wedge \mathrm{agreed}(m') \wedge$
   $(\nexists b : b < c \wedge t_b = \mathbf{vs\_trans\_sig}(p) \wedge \mathrm{vs\_vid}(t_b, p) = \mathrm{vs\_vid}(t_c, p)) \Rightarrow$
   $\forall i \forall q \forall m : t_i = \mathbf{vs\_deliver}(q, m) \wedge \mathrm{vs\_vid}(t_i, q) = \mathrm{vs\_vid}(t_c, p) \wedge \mathrm{ord}(m) < \mathrm{ord}(m'); \exists a :$
   $t_a = \mathbf{vs\_deliver}(p, m)$

Proof: The algorithm generates one **vs_trans_sig** event per view in only three cases: (1) an **evs_view** event occurs which removes one or more members from the process’ Vs_Survivors set, or after an **evs_trans_sig** event the algorithm (2) subsequently **vs_deliver**s an agreed message or (3) an **evs_view** event occurs. The guarantees provided by EVS Property 2.14 (Agreed Messages) directly apply up to either the first **evs_trans_sig** event in a VS view or first **evs_view** event that removes members from a process’ Vs_Survivors set. Delaying the **vs_trans_sig** event caused by an **evs_trans_sig** event until either VS delivering an Agreed message or a subsequent **evs_view** event does not affect the guarantees provided by that property. This is because no Agreed messages are being delivered before the VS transitional signal that were after the EVS transitional signal.

4. *If a process p delivers an agreed message m’ after a* **vs_trans_sig** *event in its current view, then p delivers every message with a lower ord value than m’ sent by all processes in p’s next transitional set that were delivered in that view. Formally:*

   $t_a= \mathbf{vs\_trans\_sig}(p) \wedge t_c= \mathbf{vs\_deliver}(p, m') \wedge t_d = \mathbf{vs\_view}(p, id', D', T') \wedge a < c < d \wedge$
   $\mathrm{agreed}(m') \wedge \mathrm{vs\_vid}(t_a, p) = \mathrm{vs\_vid}(t_c, p) = \mathrm{vs\_vid}(t_d, p) \Rightarrow$
   $\forall i \forall q \in T' \forall m \forall l \forall p' : t_i = \mathbf{vs\_send}(q, m) \wedge t_l = \mathbf{vs\_deliver}(q', m) \wedge$
   $\mathrm{vs\_vid}(t_l, q') = \mathrm{vs\_vid}(t_c, p) \wedge \mathrm{ord}(m) < \mathrm{ord}(m'); \exists b : t_b = \mathbf{vs\_deliver}(p, m)$

Proof: A process' transitional set for a VS view id' is its Vs_Survivors set upon installing that view. The fact that p installs id' implies that it received a flush message marked appropriately from each of the members of its Vs_Survivors set. This implies that all of those members installed the EVS view id'. Therefore, because they all installed EVS view id' all of those members were virtually synchronous with one another in the EVS views that they installed since their previous VS view id. Furthermore, during that time their Vs_Survivors sets were virtually synchronous, therefore, they all delivered the same set of messages that they received in the EVS views they installed in the open range [id, id'). This process received FIFO flush messages from each of the members of its Vs_Survivors. Therefore, because a process cannot send messages in a VS view after flushing it and the flush message is a FIFO message, this process received and delivered all of the messages that the members of its Vs_Survivors set sent in the previous VS view.

6.18 **Theorem (VS Safe Messages)**

1. *Safe messages are agreed messages. Formally:*

   $\text{safe}(m) \equiv m.\text{type} \in \{ S \}$

Proof: The algorithm does not affect messages' types.

2. *If a process p delivers a safe message m before a* **vs_trans_sig** *event in its current view, then every member of that view delivers m, unless it crashes. Formally:*

   $t_a = \textbf{vs\_view}(p, id, D, T) \wedge t_c = \textbf{vs\_deliver}(p, m) \wedge \text{safe}(m) \wedge \text{vs\_vid}(t_c, p) = id \wedge$
   $\nexists b : a < b < c \wedge t_b = \textbf{vs\_trans\_sig}(p) \Rightarrow$
   $\forall q \in D;\ \exists i \exists D' \exists T' \exists j : t_i = \textbf{vs\_view}(q, id, D', T') \wedge$
   $(t_j = \textbf{vs\_deliver}(q, m) \vee (t_j = \textbf{crash}(q) \wedge \text{vs\_vid}(t_j, q) = id))$

Proof: If a process delivers a Safe message in a VS view id before a **vs_trans_sig** event this implies that no **evs_trans_sig** events had occurred in the EVS views at this process since installing EVS view id. It also implies that no **evs_view** events removed members from the process' Vs_Survivors set. Therefore, because no **evs_trans_sig** events have occurred yet, every member of the EVS delivery view of m will receive m or crash. Furthermore, those members that do not crash will receive m in the same EVS view id^* as p. Those processes, therefore, will have the same Vs_Survivors set as p, due to being virtually synchronous in the EVS views they installed in the range [id, id^*). Since at that point, Vs_Survivors contains the entire membership of the VS view, all of the members of that VS view that receive m will deliver m. This argument and EVS Property 2.15 (Safe Messages) prove this theorem.

3. *If a process p delivers a safe message m after a* **vs_trans_sig** *event in its current view, then every member of p's transitional set from p's next view delivers m, unless it crashes. Formally:*

   $t_a = \textbf{vs\_view}(p, id, D, T) \wedge t_b = \textbf{vs\_trans\_sig}(p) \wedge t_c = \textbf{vs\_deliver}(p, m) \wedge b < c \wedge \text{safe}(m) \wedge$
   $t_d = \textbf{vs\_view}(p, id'', D'', T'') \wedge \text{vs\_vid}(t_b, p) = \text{vs\_vid}(t_c, p) = \text{vs\_vid}(t_d, p) = id \Rightarrow$
   $\exists i \exists D' \exists T' \exists j : \forall q \in T'' : t_i = \textbf{vs\_view}(q, id, D', T') \wedge$
   $(t_j = \textbf{vs\_deliver}(q, m) \vee (t_j = \textbf{crash}(q) \wedge \text{vs\_vid}(t_j, q) = id))$

Proof: EVS Property 2.15 (Safe Messages) guarantees that a Safe message received after a transitional signal in an EVS view will be received by all of the members of the process’ transitional set of the following EVS view, unless they crash. The transitional set of a VS view is simply the intersection of all the transitional sets of EVS views that have occurred at this process since the most recent VS view was installed. Therefore, any members of a VS transitional set will receive a Safe message m that p delivers after a **vs_trans_sig** event in a VS view, unless they crash. Furthermore, those members in Vs_Survivors that do not crash receive m in the same EVS view id* as p. Therefore, because p and its Vs_Survivors that do not crash all install that EVS view and they were in each others transitional sets, they were virtually synchronous in the EVS views that they installed in the range [id, id*). Therefore, at the point of receiving m they have the same Vs_Survivors sets and will all deliver m because p does. This argument and EVS Property 2.15 (Safe Messages) prove this theorem.

7 VS Algorithm Variants

The algorithm presented in this work used one round of n-to-n communication using FIFO messages to install views. This work also explored very similar algorithms that use more rounds and more powerful message types in order to achieve even more powerful semantics than the described VS model.

7.1 Single Round VS Algorithm Using Agreed Messages

This algorithm is almost an exact duplicate of the algorithm presented in this paper. The only difference, in fact, is that this algorithm uses Agreed messages for its flush messages and it does not install obsolete views at all. When a process receives the last necessary flush message in order to install a VS view, it only installs that view if a transitional signal has not yet been delivered in its current EVS view. If a signal was delivered, then the process “waits” for the following EVS view to be installed and then tries to install that view. Property 2.16 (Transitional Signals) guarantees that virtually synchronous processes will deliver a transitional signal at that same point in the stream of agreed messages in the view. Therefore, if one process decides not to install a view because of a transitional signal, then all of the processes that remain virtually synchronous to that process will also decide not to install that view.

The heavy additional cost of using Agreed messages instead of inexpensive FIFO messages almost surely outweighs the potential benefit of not installing obsolete views. As described in section 3, the probability that the presented algorithm actually installs obsolete views is very small.

7.2 Single Round VS Algorithm Using Safe Messages

If flush messages are Safe messages, then the presented algorithm provides a stronger set of semantics than those presented in the VS model. This variant uses the same heuristic for avoiding obsolete views as the variant that uses Agreed messages. Therefore, if a process installs a VS view, then it received a Safe flush message from each of the potential members of that VS view before any transitional signal in its current EVS view. From the properties of Safe messages, this implies that all of the other processes in the EVS view will deliver the same flush messages in the same EVS view, unless they crash. Therefore, the other processes will also receive all of the messages delivered in that EVS view before the flush messages, unless they crash first. This implies that when a process installs a VS view, all of the members of its transitional set will deliver at least the same set of messages that this process delivered in its previous VS view, unless they crash. These semantics are a stronger form of Property 2.10 (Virtual Synchrony).

7.3 Two Round VS Algorithm Using FIFO Messages

This algorithm is very similar to the single round algorithm presented in this thesis, except that this algorithm uses two rounds of flush messages instead of just one. In this variant, the first round of collecting flush messages is conducted exactly as it is in the single round FIFO variant. Once a process collects a flush message from each of the potential members of a VS view, it sends another message indicating that it has received all of the flush messages for that VS view. It then tries to collect one of these messages from each of the potential members of the VS view. If it succeeds, then the process installs the new VS view. If, instead, an EVS view is installed before it can collect the necessary messages, then the algorithm starts over and tries to install that new EVS view.

This variant provides an even stronger form of Property 2.10 (Virtual Synchrony) than the single round Safe variant provides. This is because the algorithm delivers messages as it receives them. Therefore, when a process sends the second flush message for a VS view, it has already delivered all of the messages it will deliver in its previous view. This implies that if a process installs a VS view, then all of the processes in its transitional set for that view already delivered the same set of messages that it did in its previous view. Of course, this property is not that much more powerful than the form of virtual synchrony offered by the one round safe variant, because it does not imply that the process to which the messages were delivered processed those messages before crashing. However, the virtual synchrony that this algorithm provides can be strengthened even further with a little added interaction with the client process.

In this modified variant, once the algorithm collects all the necessary flush messages in the first round, rather than immediately responding with its second flush message, it delivers another signal to the client. The algorithm only sends the second flush message when the client responds to this signal. As described above, after a process collects all of the flush messages for installing a new VS view, it has already delivered all of the messages that it will in its previous view. Therefore, if the client process “handles/processes” all of the messages delivered in its previous view before authorizing the second flush message, a very powerful form of virtual synchrony is achieved. When a process installs a view, the members in its transitional set delivered and handled all of the messages that this process did in its previous view. In effect, this algorithm implements a client-level implicit end-to-end acknowledgment of the messages that were delivered in its previous view.

The additional overhead of these variants is an additional round of n-to-n communication. However, the additional synchrony gained may or may not warrant paying that additional cost in the common case depending upon the application at hand. The performance differences between these variants and the Single Round Safe variant would be minimal, as this algorithm is effectively using “manual” safe messages.

7.4 Single Round VS Algorithm Using FIFO Messages for Spread

This work’s original purpose was to implement VS semantics on top of the Spread Wide Area Group Communication Toolkit [Spread]. Spread’s EVS semantics differ slightly from the model presented in this paper. In particular, Spread’s non-Causal messages do not maintain Same View Delivery with respect to lightweight client views. In addition, in rare cases Spread does not maintain client-level Same View Delivery for Causal messages during heavyweight daemon partitions.

Spread does not force FIFO and Reliable messages to be ordered with respect to lightweight membership changes because this would increase the cost of those messages and force them to be delivered almost with the same latency as Agreed messages. This is because a FIFO or Reliable message could not be delivered if there were any holes in the global total order of messages in the daemon's heavyweight view, because that missing message might be a lightweight membership message. Forcing this type of delivery upon FIFO and Reliable messages would drastically increase the latency of these normally loosely constrained messages. Therefore, in order to maintain the expected low latency characteristics of these messages, Spread allows them to be delivered by different daemons in different lightweight views. Spread's non-Causal messages also do not maintain Property 2.9 (Sane View Delivery 3).

Spread also violates Same View Delivery for Causal messages when daemons partition away from one another. In this case, virtually synchronous processes will deliver Causal messages in the same view, but in another network component those same messages may be delivered in different lightweight client views. This stems directly from the fact that lightweight client joins and leaves are implemented as Agreed messages. After a transitional signal in a view, different network components may deliver different sets of Safe messages. Therefore, different components may disagree upon the lightweight views in which Causal messages are delivered.

The algorithm presented in this thesis depended strongly on the fact that its flush messages be delivered in the same EVS view by virtually synchronous processes. Therefore, in this modified model, the algorithms presented, thus far, must use Causal instead of FIFO flush messages. However, this work also developed a single round FIFO algorithm in this relaxed environment where FIFO flush messages will not necessarily be delivered in the same view at all virtually synchronous processes.

The main difference between this algorithm and the earlier algorithms is that a process cannot always immediately abandon installing a VS view when a new EVS view is installed. This is because the flush messages can be delivered in different views at different processes that are virtually synchronous with one another. This could cause virtually synchronous clients to come to different decisions on whether or not to install a particular VS view. Allowing virtually synchronous processes to disagree upon which VS views to install and still attempting to meet the various GCS properties would horribly complicate any VS algorithm.

To avoid these complications, instead of abandoning VS views in response to every EVS view, this algorithm only abandons VS views when it knows that any virtually synchronous process would also abandon the VS view. The heuristic used to accomplish this is to only abandon installing a VS view if a later EVS view indicates that one of the potential members of the VS view has "gone away" before this process received its flush message for that VS view. In this case, none of the processes that are virtually synchronous to this process will deliver its flush message before seeing the same EVS view. This is because the leaving process either crashed or there was a daemon partition. In the first case, the process' flush message cannot be delivered after the membership removing it from the group. In the second case, the heavyweight daemon membership ensures that no virtually synchronous process will deliver the flush message before the view in which the process was partitioned away.

These heuristics for abandoning views, however, causes other complications. Now a process can have multiple pending VS views that it needs to try and serially install – one for each EVS view it has not yet abandoned. Furthermore, when it is forced to abandon a view because of a process leaves before its flush messages is received, that leave may force it to abandon several of its queued EVS views. In this case, several EVS view events may need to be collapsed into one aggregate VS view event.

This algorithm obviously has the drawback of installing obsolete views, because it continues to try and install views that it knows do not reflect the current underlying connectivity. This algorithm also has the problem of theoretically requiring infinite memory. If a process receives a flush message marked with an identifier that it has not seen yet, then it cannot drop that flush message. This is because that flush message could be for an EVS view that the EVS system has not yet installed at this process. On the other hand, the message could be for an EVS view that was installed in a different network component that then merged with this process' network component. There is no way for a process to implicitly differentiate between these two cases in Spread and, therefore, it must buffer the message. If that EVS view is subsequently installed at the process it can only abandon that view by the heuristic described above and it needs all of the flush messages it receives to act correctly. In Spread's EVS model, a FIFO flush message for an EVS view can, theoretically, be delivered at a process before an arbitrary number of intervening lightweight EVS views. Therefore, a process, theoretically, cannot drop any of these questionable flush messages and would require infinite memory. In a practical system, a FIFO flush message cannot be delivered before an arbitrary number of lightweight view changes. Spread uses a threshold that puts a constant limit on how "far apart" in the global ordering the same message may be delivered at different daemons. Using this threshold, this algorithm for Spread can distinguish flush messages that were meant for memberships that have already occurred and drop them.

As far as performance, the tradeoffs between this algorithm and the Single Round Causal algorithm probably cancel each other out or weigh towards the Causal Algorithm. The FIFO algorithm will have lower latency per view that it installs, but in an active group where members are commonly joining and leaving the Causal Algorithm would probably outperform it due to the smaller number of views it would install and because client applications need to authorize each and every view and client processes tend to be less responsive than if the process was under the control of the VS algorithm.

7.5 Eliminating Unnecessary Data Overhead

One drawback that all of the presented algorithms have is that they mark every message with the identifier of the VS view in which it was sent. This is a small data overhead on every message. In fact, the algorithms can be modified to eliminate the need to mark every message. As was shown in the proofs of the VS algorithm's correctness, a Causal message sent in a VS view will not be received by members of that view until after they install that view. Therefore, causal messages received from a member of a process' Vs_Survivors set were sent in its current VS view. Causal messages therefore never need to be marked and the presented algorithm marks them only for the ease of some of the proofs. Non-Causal messages do, however, need to be marked at certain times. When a non-Causal message could be received by a member of the VS view in which it was sent, before that member installs that VS view, it must be marked with proper identification so that the recipient knows whether or not it should buffer that message. Once every potential member has installed the VS view then it is safe to stop marking non-Causal messages sent in that VS view. In order to have this knowledge an n-to-n round of communication must occur in which each process announces that it has installed the view. This round of communication can proceed in parallel with sending messages in the VS view and can even be done by piggybacking on regular communication.

8 Performance

The original purpose of this work was to develop a simple algorithm that could implement partitionable Virtual Synchrony on top of Spread, which provides Extended Virtual Synchrony, with minimal impact on performance. For concrete implementation, I chose the most complex variant discussed in the previous section: one round of n-to-n FIFO[10] messages to install a membership, followed by another round of n-to-n FIFO messages to allow non-Causal messages not to be tagged with view identifiers. This implementation is known as Flush Spread.

All of the algorithms developed by this work would only incur significant overhead on top of Spread in response to group view changes. Therefore, to discover how much overhead this algorithm incurred on top of Spread the most important aspect of the GCSs to evaluate is their client-level latency in installing lightweight view changes.

The systems' lightweight view installation latencies were measured using two different membership change profiles. In both profiles the Spread daemons were placed under moderate load by external means. In the first profile lightweight view changes occurred intermittently (tens of milliseconds apart), whereas in the second profile they occurred serially in rapid fire, one immediately after the other. In both scenarios a base group of processes was constructed and then one process (the delta process) would join and leave the group several hundred times, timing how long each membership took.

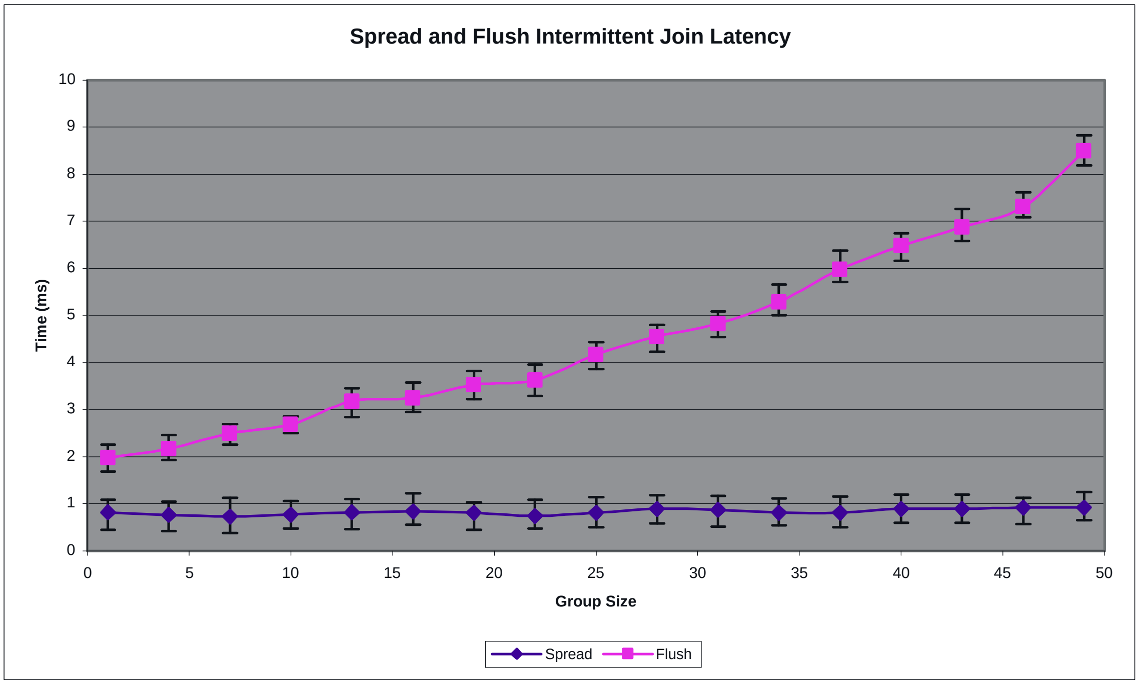
These timings were performed on a cluster of 12 dual 666MHz Pentium-IIIs with 256MB main memory interconnected over switched fast Ethernet (100Mbps). Each machine ran a single Spread daemon process and the different client processes were spread across the cluster as evenly as possible. For example, for the trials with a group size of 25 processes, there were 2 processes on 11 machines and 3 processes on 1 machine.

In Spread, the latency for join and leave lightweight view changes should be completely symmetric at all clients, as both view change types consist of sending and delivering a single Agreed message. In Flush Spread, due to an engineering decision, leaving processes do not take part in the VS algorithm. Therefore, measuring leave times at the leaving process would give false results. As it is difficult to measure the full leave latency at a process other than the leaving process, I have excluded leave membership timings from this comparison. The timings should be extremely similar to the Flush join latencies as the non-leaving members execute the exact same algorithm as they do for join view changes.

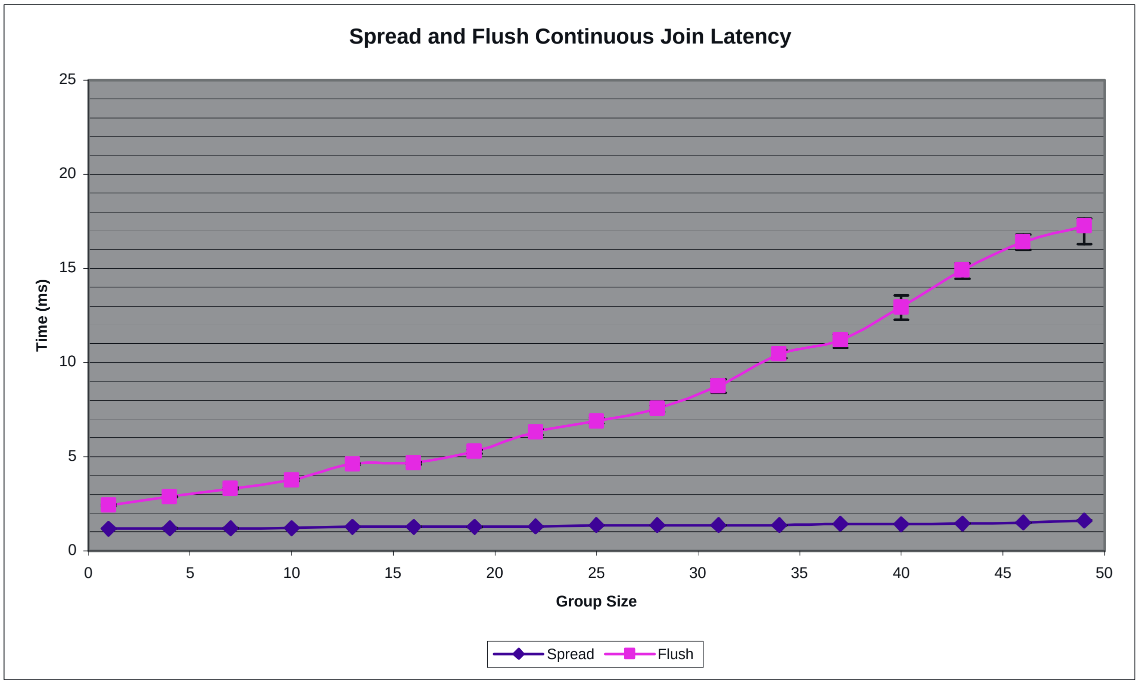
The graphs below show the median latencies it took for the views to be installed in response to a user request. The error bars on the medians represent the first and third quartile values of the latency timings.

[10] Recall that Spread does not provide Same View Delivery semantics for these FIFO messages.

These join latency timings were taken under the first membership profile of intermit tent lightweight view changes.



These join latency timings were taken under the second membership profile of rapid serial lightweight view changes.



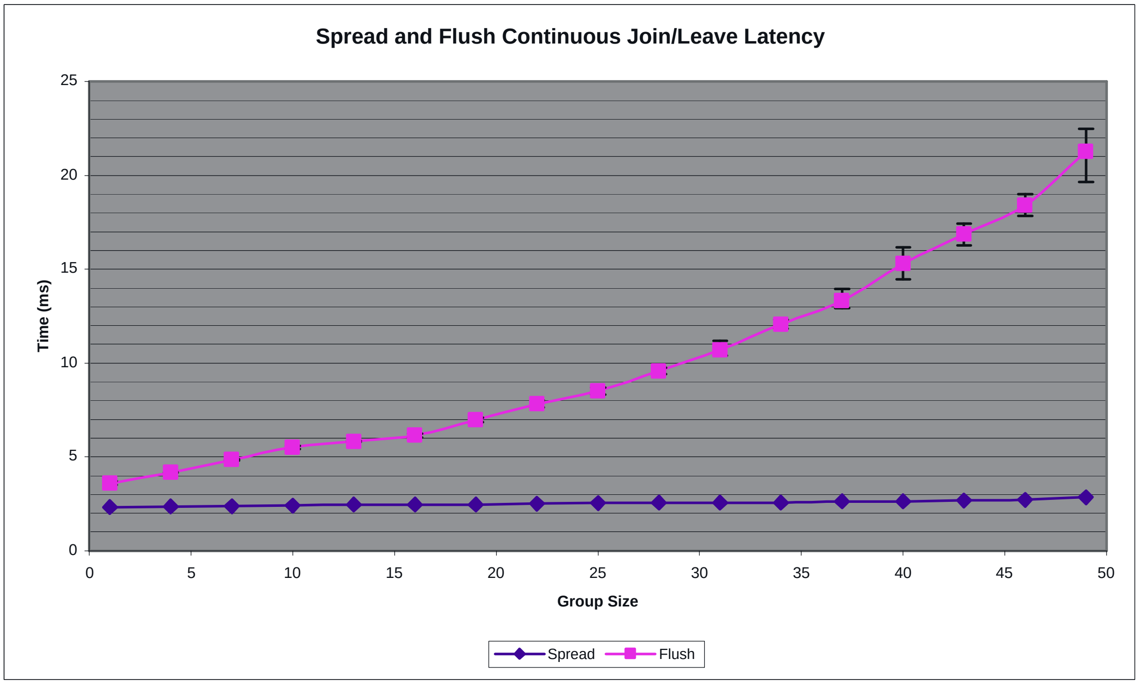
As expected, Spread's view installation latencies were almost completely unaffected as the number of client processes increased. This is due to Spread's client-daemon architecture, where the latency to deliver a message is proportional to the number of participating daemons. Once a message is deliverable a Spread daemon must simply multiplex that message to however many interested clients are connected to it. That multiplexing happens roughly in parallel and with very little additional computational load, thus making Spread relatively insensitive to the size of a process group for delivering messages. Since lightweight view changes in Spread are caused by a single Agreed message, Spread's latency to install lightweight view changes scales very well with the size of a process group.

As expected, Flush Spread's view installation latencies scaled roughly linearly as the number of client processes increased. This is due to the fact that each client must process a message from each of the n members of a potential view before installing that view. In fact, as the number of processes increased Flush Spread's latency began to become super-linear. This is due to the fact that as linearly more processes are connected to a particular Spread daemon it must deliver quadratically more total messages to those processes.

The main differences from the intermittent to the continuous scenarios are a modest increase in the median latencies and a dramatic decrease in the variability of the timings.

Spread's increase in median latency and the decrease in timing variability stem directly from Spread's local-area token Ring protocol [AMMS$^+$95, AS98] and the fact that each membership change request follows immediately after the previous view is installed. In the continuous scenario, every time the token makes one trip around each of the daemons the previous membership change request is installed. This makes the installation latency equal to the period of one token circulation and on a switched local network this period will not vary much. In the intermittent scenario the token is on average half a circulation away but could be about to arrive or have just left.

Flush Spread's large increase in latency in the continuous scenario is due to the asymmetry of join and leave events at the leaver mentioned earlier. What is happening is that when the delta process requests to leave, his request is granted at the speed of a Spread leave request and he then immediately requests to re-join the group. Since the non-leaving members are still handling his leave view change, these "join" latency timings actually time both the join latency and a good portion of the "real" leave latency. To support this theory, I timed the combined join and leave latency of Spread and Flush under the continuous scenario:



As expected, the Spread median latency for a join followed by a leave approximately doubled, but the latency for Flush Spread barely increased. This demonstrates that the "join" latencies of Flush Spread above were measuring a good portion of the combined join and leave latency. This must be the case, because this algorithm cannot install a join view change until the previous leave view change is installed by executing the full algorithm for the leave change. So the non-leaving members would slow down the delta member and he would perceive the additional time as join latency when in fact it is an artifact of his asymmetrically fast leave.

I believe that the intermittent lightweight view change profile is the more common membership change profile in practice. The continuous join/leave is more of a worst-case scenario test for the different algorithms. From the intermittent join latency graph, Flush Spread has a latency of less than 10ms for an n-to-n round to complete between fifty participants on a fast local network. If membership changes are not too common, then Flush Spread could comfortably support groups of hundreds of processes in that environment. This is almost an order of magnitude slower than Spread and scales much worse as the group size increases. However, one of the main claims of this paper was that Spread's architecture was of such high performance that a more powerful and expensive set of GCS semantics could be implemented on top of it without excessive overhead.

9 Conclusions

This thesis presented several distributed algorithms for implementing the Virtual Synchrony (VS) model of group communication on top of the Extended Virtual Synchrony (EVS) model of group communication. It formally proved that a more powerful set of GCS semantics could be built on top of a weaker set of semantics with very simple algorithms. Furthermore, this thesis argued that, in the common case, these algorithms have competitive performance compared with other “native” implementations of the VS model.

10 References

[ACBMT95] E. Anceaume, B. Charron-Bost, P. Minet and S. Toueg. On the formal specification of group membership services. TR 95-1534, Dept. of Computer Science, Cornell University, August 1995.

[ACDV97] Y. Amir, G. V.Chokler, D. Dolev and R. Vitenberg. Efficient state transfer in partitionable environments. In *2nd European Research Seminar on Advances in Distributed Systems (ERSADS'97)*, pages 183-192. BROADCAST (ESPRIT WG 22455), Operating Systems Laboratory, Swiss Federal Institute of Technology, Lausanne, March 1997. Full version available as Technical Report CS98-12, Institute of Computer Science, The Hebrew University, Jerusalem, Israel.

[ADKM92] Y. Amir, D. Dolev, S. Kramer and D. Malki. Transis: A communication sub-system for high availability. In *22nd IEEE Fault-Tolerant Computing Symposium (FTCS)*, July 1992.

[Ami95] Y. Amir: *Replication Using Group Communication Over a Partitioned Network*. Ph.D. Thesis, Institute of Computer Science, The Hebrew University, Jerusalem, Israel, 1995.

[AMMS+95] Y. Amir, L.E. Moser, P.M. Melliar-Smith, D.A. Agarwal and P. Ciarfella. The totem single-ring ordering and membership protocol. *ACM Trasactions on Computer Systems*, 13(4):311-342, November 1995.

[AS98] Y. Amir and J. Stanton. The Spread Wide Area Group Communication System. TR CNDS-98-4, The Center for Networking and Distributed Systems, The Johns Hopkins University, 1998.

[Bir86] K. Birman. ISIS: A System for Fault-Tolerant Distributed Computing. Technical Report TR86-744, Cornell University, Department of Computer Science, April 1986.

[BvR94] K. Birman and R. van Renesse. *Reliable Distributed Computing with the Isis Toolkit*. IEEE Computer Society Press, 1994.

[CHTCB96] T. Chandra, V. Hadzilacos, S. Toueg and B. Charron-Bost. On the impossibility of group membership. In *15th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 322-330, May 1996.

[DPFLS98] R. Prisco, A. Fekete, N. Lynch and A. Shvartsman. A dynamic view-oriented group communication service. In *17th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 227-236, June 1998.

[FvR95] R. Friedman and R. van Renesse. Strong and Weak Virtual Synchrony in Horus. TR 95-1537, Dept. of Computer Science, Cornell University, August 1995.

[GG91] S. Garland and J. Guttag. A guide to LP, the Larch P rover. Research Report 82, Digital Systems Research Center, 130 Lytton Avenue, Palo Alto, CA 94301, December 1991.

[GHG+93] J. Guttag, J. Horning, S. Garland, K. Jones, A. Modet and J. Wing, editors. *Larch: Languages and Tools for Formal Specification*. Sp ring-Verlag Texts and Monographs in Computer Science, 1993.

[GL98] S. Garland and N. Lynch. The IOA language and toolset: Support for designing, analyzing and building distributed systems. Technical Report MIT/LCS/TR-762, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, August 1998. URL http://theory.lcs.mit.edu/tds/papers/Lynch/IOA-TR-762.ps

[GS95] R. Guerraoui and A. Schiper. Transaction model vs virtual synchrony model: bridging the gap. In *Theory and Practice in Distributed Systems*, LNCS 938, pages 121-132. Springer-Verlag, September 1995.

[HvR96] J. Hickey, N. Lynch and R. van Renesse. Specifications and proofs for ensemble layers. In *Fifth International Conference on Tools and Algorithms for the Construction an d Analysis of Systems, (TACAS '99, Amsterdam, Netherlands, March 1999)*, Lecture Notes in Computer Science. Springer-Verlag, 1999.

[Lam78] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558-565, July 1978.

[LSGL95] V. Luchangco, E. Soylemez, S. Garland and N. Lynch. Verifying timing properties of concurrent algorithms. In Dieter Hogrefe and Sefan Leue, editors, *Formal Description Techniques VII: Proceedings of the 7th IFIP WG6.1 International Conference on Formal Description Techniques* (FORTE'94, Berne, Switzerland, October 1994), pages 259-273. Chapman and Hall, 1995.

[LT89] N. Lynch and M. Tuttle. An introduction to Input/Output Automata. *CWI Quarterly*, 2(3):219-246, 1989.

[Lyn96] N. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, 1996.

[KK00] I. Keidar and R. Khazan. A client-server approach to virtually synchronous group multicast: specifications and algorithms. In *20th IEEE*

International Conference on Distributed Computing Systems (ICDCS), pages 344-365, April 2000.

[MAMSA94] L. Moser, Y. Amir, P. Melliar-Smith and D. Agarwal. Extended virtual synchrony. In *14th International Conference on Distributed Computing Systems (ICDCS)*, pages 56-65, June 1994. Full version: technical report ECE93022, Department of Electrical and Computer Engineering, University of California, Santa Barbara, CA.

[MPS91] S. Mishra, L. Peterson and R. Schlicting. Consul: A Communication Substrate for Fault-Tolerant Distributed Programs. TR 91-32, Dept. of Computer Science, University of Arizona, 1991.

[PPG$^+$96] T. Petrov, A. Pogosyants, S. Garland, V. Luchangco and N. Lynch. Computer-assisted verification of an algorithm for concurrent timestamps. In Reinhard Gotzhein and Jan Bredereke, editors, *Formal Description Techniques IX: Theory, Applications and Tools* (FORTE/PSTV'96: Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols, and Protocol Specification, Testing and Verification, Kaiserslautern, Germany, October 1996), pages 29-44. Chapman & Hall, 1996.

[RKM96] R. V. Renesse, K. Birman and S. Maffei. Horus: A flexible group communication system. *Communications of the ACM*, 39:76-83, April 1996.

[SAGG$^+$93] J. Sogaard-Andersen, S. Garland, J. Guttag, N. Lynch and A. Pogosyants. Computer-assisted simulation proofs. In Costas Courcoubetis, editor, *Computer-Aided Verification* (5th International Conference, CAV'93, Elounda, Greece, June/July 1993), volume 697 of *Lecture Notes in Computer Science*, pages 305-319. Springer-Verlag, 1993.

[Spread] The Spread Wide Area Group Communication Toolkit. http://www.spread.org

[Vit98] R. Vitenberg. Properties of distributed group communication and their utilization. Master's thesis, Institute of Computer Science, The Hebrew University of Jerusalem, Jerusalem, Israel, January 1998.

[VKCD99] R. Vitenberg, I. Keidar, G. Chokler and D. Dolev. Group Communication Specifications: A Comprehensive Study. Technical report CS99-31, Computer Science Institute, The Hebrew University, Jerusalem, Israel. MIT Technical Report MIT-LCS-TR-790, September 1999.

[WMK94] B. Whetten, T. Montgomery and S. Kaplan. A high performance totally ordered multicast protocol. In *Theory and Practice in Distributed Systems*, *International Workshop*, Lecture Notes in Computer Science, page 938, September 1994.

[WS95] U. Wilhelm and A. Schiper. A hierarchy of totally ordered multicasts. In *14th IEEE International Symposium on Reliable Distributed Systems (SRDS)*, September 1995.