Strong and Weak Virtual Synchrony in Horus*

Roy Friedman Robbert van Renesse

Department of Computer Science
Cornell University
Ithaca, NY 14853.

August 24, 1995

Abstract

A formal definition of *strong virtual synchrony*, capturing the semantics of virtual synchrony as implemented in Horus, is presented. This definition has the nice property that every message is delivered within the view in which it was sent. However, it is shown that in order to implement strong virtual synchrony, the application program has to block messages during view changes.

An alternative definition, called *weak virtual synchrony*, which can be implemented without blocking messages, is then presented. This definition still guarantees that messages will be delivered within the view in which they were sent, only that it uses a slightly weaker notion of what the view in which a message was sent is. An implementation of weak virtual synchrony that does not block messages during view changes is developed, and it is shown how to use a system that provides weak virtual synchrony even when strong virtual synchrony is actually needed.

To capture additional ordering requirements, the definition of *ordered virtual synchrony* is presented. Finally, it is discussed how to extend the definitions in order to cope with the fact that a process can become a member of more than one group.

*This work was supported by ARPA/ONR grant N00014-92-J-1866

1 Introduction

Virtual synchrony is a convenient paradigm for developing distributed applications in asynchronous systems in which processes may crash, messages may get lost, and the communication network may get partitioned, since it *simulates* a reliable delivery fail-stop model to the application. That is, virtual synchrony creates an illusion to the application that it runs in an environment in which crashed processes are always detected, and if a certain process is suspected of being crashed, then this process has really crashed. This is done by presenting processes with *views*, which consists of the set of currently reachable and operational processes. The system then guarantees that between every two consecutive views v_1 and v_2, no message that was sent from a process not in v_1 can be delivered and that all processes that appear in both v_1 and v_2 have to see the same set of messages.

In particular, the use of virtually synchronous communication systems greatly simplifies the task of developing replicated services: It it possible to send a message to the entire set of processes, and the system would ensure that all live replicas will receive a copy of the message. Moreover, if one of the replicas would become faulty, other replicas will learn about it by receiving a new view which does not include the crashed replica, and one of the live replicas would take over the job of the crashed replica. Also, in the event of a network partition, each partition will receive a view which includes only the processes that belong to that partition. Later, when the communication links are fixed, a specific join request would allow the two partitions to negotiate a new state. Finally, after the combined state of the two partitions has been resolved, a new view which consists of all processes would be generated for all participating processes.

During the course of years, several systems that support a virtually synchronous communication paradigm have been developed [1, 2, 3, 5, 25]. These systems usually perform quite well in terms of average message latencies, but exhibit harsh slow downs during view changes, making them impractical for applications that require timed response. In this paper, we report a formal and practical study of the membership layer of Horus [24], i.e., the layer which is responsible for implementing virtual synchrony, aimed at improving this situation. That is, we would like to come up with a membership layer that will be very efficient in the normal case, and will continue to perform well during view changes.

We start by presenting a formal definition of *strong virtual synchrony*, capturing the semantics of virtual synchrony as it is implemented by Horus. This is the first time that a formal definition of the membership layer of Horus is given. The definition of strong virtual synchrony includes a requirement that a message is delivered only within the view in which it was sent. The only other definition to include this requirement is the definition of *extended virtual synchrony*, which is supported by the Transis project [1, 2] and the Totem project [3], but is *not* supported by other definitions of virtual synchrony, including the definition used by ISIS [5, 7].

We believe that this additional requirement is important for many applications, since it allows for immediate local delivery of messages, without any further computations or bookkeeping. In other systems that do not have this guarantee [4, 5, 7, 9], local deliveries must be delayed until the system can figure out in what view the message is supposed to be delivered. Also, the fact that messages are delivered within the view in which they were sent can reduce the amount of information that needs to be sent with each message, and simplifies the computation which is needed in order to handle each message. For example, the CAUSAL and ORDER layers of Horus implements causal [15] and uniform [13, 20] delivery of messages by using vector timestamps. The fact that the view of the sender when a message is sent is identical to the views of the receivers when the message is delivered, allows the sender to send only a local vector timestamp without any additional context information. The recipients of the message can safely assume that the ith entry of the vector they received corresponds to the ith member in their view.

The current implementation of the membership layer of Horus blocks messages during view changes. In this paper, we show that this blocking is necessary in order to support strong virtual synchrony. Specifically, we show a lower bound, stating that in every implementation of strong virtual synchrony, there exists a time d such that messages cannot be sent at least d units of time before a view change. Here, d is the message delay of the underlying system (layers), including the network itself.

The lower bound that we show depends on the requirement that messages are always delivered within the view in which they are sent. On the other hand, in other systems that do not support this requirement, local deliveries must incur some delays and possibly additional bookkeeping, while some applications become more complex due to the need to add more context information to every message. The immediate question that comes up is if this is a real trade-off, or can we enjoy the benefits of both worlds? In this paper we give a partial answer to this question by presenting the definition of *weak virtual synchrony*, which allows for immediate local deliveries, but does not require blocking messages during view changes and adds only little additional header information to messages. According to the definition of weak virtual synchrony, during view changes, processes are supplied with a temporary *suggested view*. Processes can send messages in a suggested view, and are guaranteed that this message will be delivered within the next real view, and that the next real view will be an ordered subset of the suggested view. We explain below the usefulness of these guarantees in reducing the context information that needs to be sent when compared with other definitions that allow to continue sending messages during view changes.

We have developed a protocol that provides weak virtual synchrony and does not block messages during view changes. In the actual implementation of this protocol in Horus, during views and suggested views, processes receive a membership list and an indication which of the processes that appear in the membership list have failed. However, due to the guarantees of suggested views, the membership list which is passed to the application is only changed in the first suggested view event after a real view event and in real view events. In other suggested view events, the membership list remains the same, and only the indication about which processes have failed may change. Hence, for most messages, the membership list is the same both when they are sent and when they are received. The only two cases where this may not hold are the following: (a) a message that was sent between a suggested view event and the next view event, but was received after the view event, and (b) a message that was sent when a view is installed, but is received after a following suggested view event. For these messages, the application can maintain a translation table, between the ranks of the members in the old list and their ranks in the current list. We have used this technique in order to extend many existing layers of Horus to work with weak virtual synchrony, which turned out to be a very simple task.

Of course, the same thing can be done with other definitions of virtual synchrony. However, with weak virtual synchrony, the application needs to maintain at most two translation tables at a time. In order to do the same with other definitions of virtual synchrony that allow to send messages during view changes, the application may need to maintain an arbitrary large number of translation tables simultaneously. And, of course, other definitions of virtual synchrony that allow to send messages during view changes do not allow for immediate local deliveries.

We have compared the latency of messages sent during view changes in both the implementation of strong virtual synchrony and the implementation of weak virtual synchrony with the latency of messages sent during normal operation. Our measurements indicate that messages sent during view changes in the weakly virtually synchronous implementation are slower than regular messages only by a small constant factor that does not depend on the load of the system. On the other hand, messages that were sent during view changes in the strongly virtually synchronous implementation were significantly slower than those sent in the weakly virtually synchronous implementation, and this gap becomes larger as the load on the system increases.

Our definitions of weak and strong virtual synchrony do not impose any ordering restrictions on messages sent within the same view. However, in many distributed applications, it is useful to have at least some of the messages delivered either in a total order or in a causal order. To capture these ordering requirements, we add the definition of *ordered virtual synchrony* and a short discussion about how these requirements are implemented in Horus.

Finally, we discuss how to extend our definitions so they can cope with the fact that a process may become a member of more than one group. We introduce the notion of *failure domains*, which means that if a process becomes a suspect in one group, it must be declared suspect in all the groups in which it is a member. We suggest a formal definition for failure domains, although this definition is slightly weaker than the semantics provided by Horus in the case that a process can be suspected of being faulty more than once.

The rest of this paper is organized as follows: Related work is discussed in Section 2. We present the formal model in Section 3. The definition of strong virtual synchrony and the lower bound on its implementations is presented in Section 4. The definition of weak virtual synchrony is presented in Section 5 while the protocol for implementing weak virtual synchrony together with the latency measurements are given in Section 6. In Section 7 we present the definition of ordered virtual synchrony. Multiple groups and failure domains are discussed in Section 8 and we conclude with a discussion in Section 9.

2 Related Work

The notion of virtual synchrony was first introduced by Ken Birman in the ISIS project [5, 8, 7]. However, this definition of virtual synchrony in ISIS is somewhat different than our definitions, as it does not provide any guarantee on the view in which a message will be delivered. Also, the definition of virtual synchrony in ISIS requires the existence of a primary partition. Having a primary partition is important for applications in order to acquire locks, or to perform any kind of operations which require coherent behavior, e.g., directing an airplane to a certain area in the sky. Our definitions allow keeping track of a primary partition, given some reasonable rules for deciding if a certain partition is primary or not. An example of such a rule can be the majority of processes within a fixed group of processes. On the other hand, our definition does not require to have only one primary partition at all times, so applications that can safely make progress even in the absence of a primary partition, will be allowed do so. As in our definitions, virtual synchrony in ISIS does not require *uniformity* [20, 13] of regular messages (also called *safe delivery* in [3]), i.e., that if a message is received by any process, faulty or non-faulty, then every non-faulty process must receive it too. The decision not to support uniformity of messages is motivated by the high cost associated with providing this guarantee. However, in Horus, uniformity of messages can be added by using the ORDER layer [24].

Throughout the years, several other definitions of virtual synchrony were introduced, at different levels of formality, and several algorithms for implementing them were also developed. These works include projects like Transis [1, 2], Totem [3], Relacs [4], and Newtop [12], as well as several other papers like [9, 10, 23]. [1] Some of these works, e.g., [1, 2, 3, 10, 23], include a variant of virtual synchrony which requires uniformity of messages, but others, e.g., [4, 9, 12], do not. The only other definition that include the requirement that a message is always delivered within the view in which it was sent is the definition of extended virtual synchrony which is supported by Transis [1, 2] and Totem [3]. However, except for the Relacs protocol [4], and the Newtop protocol [12], all of the protocols for providing virtual synchrony developed in these works stop sending messages during the installation of new views, sometimes called the *reformation phase*. This blocking of messages might be inherent in the protocols themselves, but is not required by the semantics that these protocols provide. (We want to make clear that there is a difference between what the protocol does internally, and what the semantics that this protocols presents to the application is. In particular, it is possible that if some of these protocols would have added a flush event to notify the application that a configuration change is taking place, the resulting semantics would have been similar to strong virtual synchrony. However, the fact is that these protocols do not inform the application when they block its messages, so the application cannot take advantage of this.)

The definition of extended virtual synchrony [3] includes the notion of transitional view, which must be delivered before a real view can be installed. The difference between a suggested view and a transitional view is that a suggested view is a superset of the next real view, and

[1] The work of Chang and Maxemchuk [10] does not use the term virtual synchrony, but defines a very similar semantics.

is used to allow processes to continue sending messages during view changes. In contrast, a transitional view consists of the processes that appear in the intersection of the previous and next real views, and is used to notify the application that all messages sent in the previous real view are now stable w.r.t. to all the processes that appeared in that view.

One major difference between Horus and most other systems which provide virtual synchrony is that Horus is a layered system, in which different concerns are decoupled, and each concern is implemented in a different layer. The definitions of strong and weak virtual synchrony developed in this paper follow this guideline. Hence, issues like uniformity (safe delivery), causal ordering, and atomic delivery of messages are not part of these definitions, and are not implemented in the membership layer of Horus. However, there are separate layers which can add any of these requirements to the overall semantics, in order to support applications that need a stronger semantics.

We do not claim that virtual synchrony is necessary for all types of applications. For instance, applications which are very asynchronous may prefer a model in which a message that was sent to a crashed process will be delivered to this process when it is re-booted. Examples of systems that provide such a model include GTS [17], the ISIS wide-area facility [18], and the Long-Term layer of Horus. (In fact, the Long-Term layer of Horus runs on top of the membership layer, and uses the regular membership layer to propagate messages to the set of currently active processors.)

The Psync protocol [21] provides causal delivery of messages among a fixed set of processes. Psync does not support additional properties like total ordering or dynamic membership changes, but provides hooks that enable higher level applications to do so, as done in Consul [19].

Other related work is transaction-based technology, for example, such as used in the Harp file system [16]. However, because these systems use only totally ordered transactions, their performance is much worse than what can be achieved with Horus, and there are many applications, e.g., replicated servers and mission control systems, that do not need the semantics of atomic transactions. A more detailed discussion of these issues appears in [6, 14, 26].

Since the membership layer of Horus is implemented on top of layers which do not provide virtual synchrony, any potential application of Horus can also be developed directly with point-to-point communication. However, we believe that, in general, developing a reliable distributed application on top of a virtually synchronous system like Horus is simpler than without such a layer. This is because virtual synchrony shields the programmer from many of the bad scenarios that can otherwise happen in a distributed environment, resulting in fewer lines of code, simpler code, and a higher degree of confidence in the correctness of the implementation.

In a recent paper, Ricciardi and Birman have formalized the notion of group membership using temporal logic in [22]. However, this definition deals only with view changes, and does not refer to other messages and their ordering.

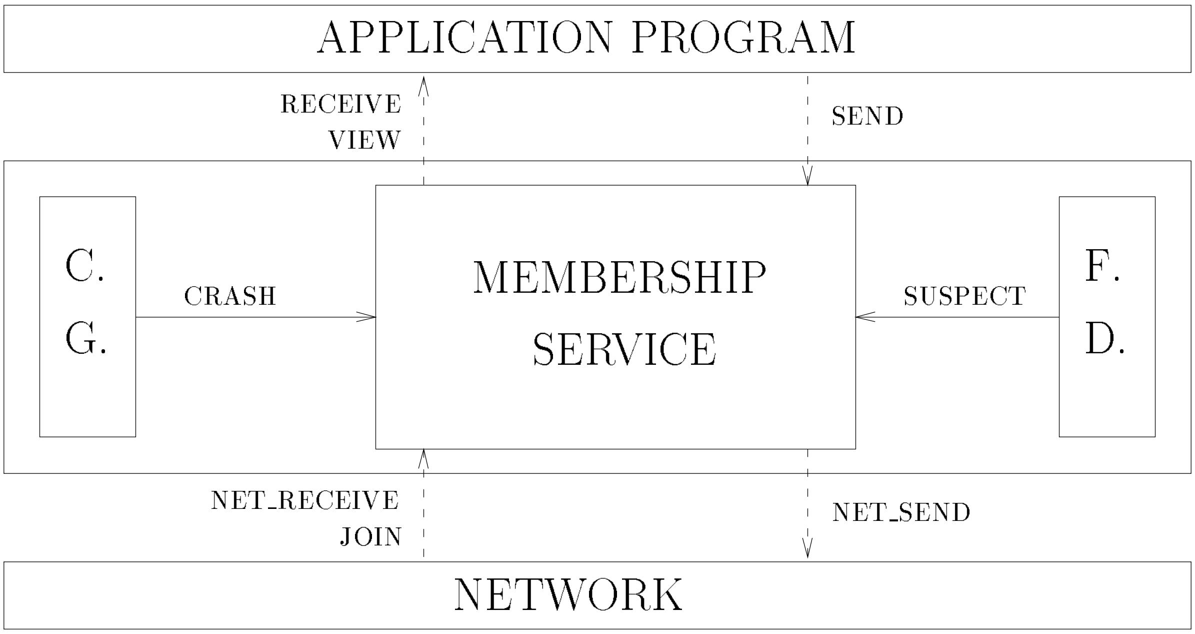

Figure 1: System illustration

3 The Model

We assume a finite (possibly unbounded) set of nodes S, connected via some interconnection network. Each node in S consists of the following components: an *application program*, a *membership service*, a *failure detector*, and a *crash generator*. The membership service is an automaton, that accepts events either from the network or from the other components of the same node, do some local computation, and generates zero or more events to the network and the other components of the same node. (See illustration in Figure 1.) The membership service can accept *send* events from the application program, *suspect* events from the failure detector, *crash* events from the crash generator, and *net_receive* and *join* events from the network; the membership service may generate *receive* and *view* events for the application program, and *net_send* events for the network. In the rest of this paper, when we say *process* we refer to the membership service.

A *history* h_i, is a sequence of events that occur in a process p_i, in the order of their occurrence:

view: a view event ev includes an ordered list of processes, denoted by $ev.view$, and a view id number, denoted by $ev.vid$.

send: a send event ev includes a message, denoted by $ev.msg$, a list of processes to whom the message should be sent, denoted by $ev.target$, the id of the process in which the event occurs, denoted by $ev.pid$, and a view id number, denoted by $ev.vid$.

receive: a receive event ev includes a message, denoted by $ev.msg$, and an id number of the process that sent the message, denoted by $ev.pid$.

join: a join event ev includes a process id, denoted by $ev.pid$.

suspect: a suspect event ev includes a list of processes, denoted by $ev.suspects$.

crash: a crash event has no parameters.

net_send: a net_send event ev includes a message to be sent, denoted by $ev.msg$, and a list of processes to whom the message should be sent, denoted by $ev.target$.

net_receive: a net_receive event ev includes a message, denoted by $ev.msg$.

We assume also that messages are unique, i.e., for every two send events ev and ev', $ev.msg \neq ev'.msg$ and if ev appears in some history h_i, then ev does not appear in any other history h_j, $j \neq i$. (This can be implemented by timestamping the messages.) Note that since a receive event and a view event are not characterized by the process in which they occurred, the same receive or view events may occur in several histories.

Given a sequence of events h and two events ev and ev' in h, we denote the fact that ev appears in h before ev' by $ev \xrightarrow{h} ev'$. Similarly, given an ordered list of processes v and two processes p_i and p_j in v, we denote the fact that p_i appears in v before p_j by $p_i \xrightarrow{v} p_j$.

We say that a process history h_i is *admissible* if the following holds:

1. If h_i includes a crash event ev, then ev is the last event in h_i.
2. For every send event ev in h_i, there exists a view event ev' such that $ev' \xrightarrow{h_i} ev$, $ev'.vid = ev.vid$, $ev.target \subseteq ev'.view$, and there does not exist another view event ev'' such that $ev' \xrightarrow{h_i} ev'' \xrightarrow{h_i} ev$.

An *execution* is a collection of process histories, one for each process. We say that an execution σ is *admissible* if every history h_i of p_i in σ is admissible and for every net_receive event ev in every history h_i in σ, there exists a net_send event ev' in $h_{ev.pid}$ such that $ev.msg = ev'.msg$ and $p_i \in ev'.target$. From now on, we assume that all executions are admissible.

We assume in this paper that the system is asynchronous. That is, different processes may run at a different rate, and the delay of messages is unknown, and may vary from one message to another. For reasons of time analysis, we assume that each event ev in every history h_i is associated with some real time $\mathcal{R}_{h_i}(ev)$. We define $\mathcal{R}_{h_i}(ev)$ to be $\perp$ if ev is not included in h_i. We emphasize that a process may not know the real time associated with the events in its history. Given a send event ev in some history h_i of an execution σ, and a receive event ev' in a history h_j of σ such that $ev.msg = ev'.msg$, we define $d_j(ev.msg) = \mathcal{R}_{h_j}(ev') - \mathcal{R}_{h_i}(ev)$; we define $d(ev.msg)$, the delay of the message, to be the maximum of $d_j(ev.msg)$ over all js for which ev' is included in h_j, and to be ∞ if ev' does not exist in any history of σ. We assume that for every send event ev, $d_j(ev.msg) > 0$.

A *protocol* or an *algorithm* is any description of the transition function of the membership service, i.e., any description of the local computations and events that should be generated by the membership service whenever it accepts an event.

Note, that we do not explicitly model the failure detector, the application program, or the communication network. However, the behavior of these elements must conform with the other requirements of the system. In particular, the network may not generate spurious messages, and may not duplicate messages. However, whether the actual network has these characteristics, or in fact the network is totally unreliable, but there is a layer which sits between the "real" network and the interface to the membership service in order to provide these guarantees, is none of our concern in this paper. This matches the approach of Horus, in which each layer implements only its own functionality, and does not care which layers are placed below and how they are implemented, as long as these lower layers provide the right semantics.

Note also that Figure 1 illustrate the system architecture as it appears in a single node. This is why the events of the crash generator are sent to the membership service and not to the failure detector. That is, a crash event generated by the crash generator only causes the local membership service to crash by halting. (In particular, crashes do not cause the membership service to generate spurious messages).

4 Strong Virtual Synchrony

In this section, we define the notion of strong virtual synchrony formally, capturing the original semantics provided by the membership layer of Horus. We then show that these semantics impose long delays on messages that are about to be sent just before a view change.

The definition of strong virtual synchrony is divided into two parts. In the first part, we define the requirements from views, while the second part includes the requirements about the ordering of messages w.r.t. these views. However, before we can introduce these definitions, we need the following notion of *consecutive view events in a history.* Formally, we say that two view events ev' and ev'' are consecutive view events in a history h_i if $ev' \xrightarrow{h_i} ev''$ and there does not exists another view event ev''' such that h_i if $ev' \xrightarrow{h_i} ev''' \xrightarrow{h_i} ev''$.

An intuitive explanation of the following definition is given in the text below it.

Definition 4.1 (Strong View Admissibility) *An execution σ is strongly view admissible if the following holds:*

1. *For every history h_i in σ and every view event ev in h_i, p_i is included in ev.*

2. *If ev and ev' are two view events in h_i such that $ev \xrightarrow{h_i} ev'$, then $ev.vid < ev'.vid$.*

3. *If some history h_i includes a crash event ev and there exists another history h_j, $j \neq i$, that includes a view event ev' such that $p_i \in ev'.view$, then there exists another event ev'' in h_j such that $ev' \xrightarrow{h_i} ev''$ and either ev'' is a view event and $p_i \notin ev''.view$ or ev'' is a crash event.*

4. *If some history h_i includes a join event ev and does not include a crash event, then h_i includes another event ev' such that $ev \xrightarrow{h_i} ev'$ and either ev' is a view event such that $ev.pid \in ev'.view$ or ev' is a suspect event and $ev.pid \in ev'.suspects$.*

5. *For every view event ev that appears in a history h_i and every process p_j such that $p_j \in ev.view$, if ev is the last view event in h_i and h_i does not include a crash event, or if ev' is a consecutive view event in h_i and $p_j \in ev.view \cap ev'.view$, then h_j includes ev.*

6. *For every two consecutive view events ev and ev' in a history h_i and every process p_j, if $p_j \in ev.view \setminus ev'.view$, then there exists a history h_k that includes a suspect event ev'' such that $p_j \in ev''.suspects$, $ev \xrightarrow{h_k} ev''$ and there is no view or crash event ordered in h_k between ev and ev''.*

The first condition in the definition of strong view admissibility requires that every process will be included in all of its local views. The second condition requires that the *vid* field of view events will reflect their relative order within a history. The third condition requires that every process that crashes will eventually be removed from the views of all processes (if it appeared there). The fourth condition requires that if a process wishes to join a view, it will either be added eventually, or declared suspect. The fifth condition requires that if process p_j appears in two consecutive views of another process p_i, denote the first one of these views by V, then p_j must have seen view V as well. The sixth condition requires that a process can only be removed from a view if it was suspected of being faulty by a member of that view. (Note that in this case, h_k is not necessarily the same as h_i and therefore, ev and ev' need not be consecutive view events in h_k.)

Before we can define strong virtual synchrony, we introduce the notion of the view event which generates the view in which a message is sent. Given a send event ev and a view event ev', we say that *ev' generates the view in which $ev.msg$ is sent* if $ev.vid = ev'.vid$.

An intuitive explanation of the definition below, appears in the text that follows.

Definition 4.2 (Strong Virtual Synchrony) *An execution σ is strongly virtually synchronous if the following holds:*

1. *σ is strongly view admissible.*

2. *For every history h_i in σ and every receive event ev in h_i, if ev' is the view event that generates the view in which $ev.msg$ is sent, then $ev' \xrightarrow{h_i} ev$, and if there exists a view event ev'' in h_i such that $ev''.vid > ev'.vid$, then $ev \xrightarrow{h_i} ev''$.*

3. *Let ev be a send event, ev' the view event in which $ev.msg$ was generated, and p_i a process such that $p_i \in ev.target$, then if ev' is the last view event in both h_i and $h_{ev.pid}$ and neither h_i nor $h_{ev.pid}$ includes a crash event, or if ev'' is a consecutive view event in both h_i and $h_{ev.pid}$, then there exists a receive event ev''' in h_i such that $ev.msg = ev'''.msg$.*

4. *Let ev be a send event, ev' the corresponding receive event, ev'' the view event in which $ev.msg$ is generated, and h_i a history that includes both ev' and ev''. Then the following holds: (a) If ev''' is a consecutive view event in h_i, and $ev' \xrightarrow{h_i} ev'''$, then every history h_j, in which ev'' and ev''' are consecutive view events, includes ev'. (b) If ev'' is the last view event in h_i and there are no crash events in h_i, then every history h_j, in which ev'' is the last view event in h_j and h_j does not include a crash event, includes ev'.*

5. *For every history h_i and two send events ev and ev' such that for some history h_j, $ev \xrightarrow{h_j} ev'$, the view in which $ev.msg$ and $ev'.msg$ are generated is the same, and $p_i \in ev.target \cap ev'.target$, if h_i includes the receive event that corresponds to ev', denoted by ev''', then h_i includes the receive event that corresponds to ev, denoted by ev'', and $ev'' \xrightarrow{h_i} ev'''$.*

By the first condition in the definition of strong virtual synchrony, a strongly virtually synchronous execution must also be a strongly view admissible execution. The second condition requires that every message must be delivered within the view in which it was sent. The third condition requires that every message sent by a process that remains in the view of other processes, is delivered by these processes. This requirements guarantees reliable delivery of messages. The fourth condition requires that all "surviving" members of a view agree on the set of messages delivered within this view. The fifth condition requires that between two consecutive view events, Horus simulates a no-omission failures model among messages sent to the same destinations, and that these messages are delivered in fifo order.

A protocol $\mathcal{P}$ implements strong virtual synchrony if every execution generated by it is strongly virtually synchronous. For the rest of this paper, we use SVS as shorthand for strong virtual synchrony.

Note that the *vid* field is used in the definitions of admissible executions, view admissibility, and virtual synchrony only for bookkeeping and these definitions could have been written without it. However, without the *vid* field these definitions would have been more complex and formal reasoning about them would have been even more difficult. Also, currently there is no starting event in the definition of admissible executions. This may contradict the intuitive thought that an application program needs to do something active in order to become a member of a group. To overcome this concern, we can introduce a new event, **start**, that the membership service may accept from the application program, and require that this event will be the first event in every admissible history. We have decided not to add this event to the formal definition of admissible executions presented in this paper, since it is not required for any of the other results presented in this paper. On the other hand, we believe that the current definitions allow to develop self stabilizing protocols [11], and by adding a starting event we would have prohibited such protocols.

The following lemma shows that in every implementation of strong virtual synchrony, processes have to stop sending messages at least d time before a new view is installed, where d is the delay of the underlying layers (including the delay of the network itself).

Lemma 4.1 (Lower Bound for SVS) *In every SVS implementation, no send event can be generated by a process p_i at least d time before a view event is generated by any process that appears in the target field of this send event.*

Proof: Assume, by way of contradiction, that there exists an SVS implementation $\mathcal{E}$ in which a send event can be generated $d' < d$ time before a view event. Consider the following execution σ of $\mathcal{E}$ that includes two processes p_0 and p_1 that send a message to every process in their view at least every $d' < d$ time. We assume that there exist two consecutive view events ev and ev' in h_0 and h_1; ev occurs at real time t_0 in h_0 and at real time t_1 in h_1; ev' occurs at real time t'_0 in h_0 and at real time t'_1 in h_1. In particular, this means that both p_0 and p_1 are included in $ev.view$ and $ev'.view$. We assume also that $t'_1, t'_0 > t_0 + d'$ and $t'_1, t'_0 > t_1 + d'$.

Assume, without loss of generality, that $t'_1 > t'_0$. Hence, during the time interval $[t'_0 - d', t'_0]$, there is a send event ev'' by p_1 such that $ev''.vid \leq ev.vid$ and $p_0 \in ev''.target$. Due to the message delay, the corresponding receive event occurs in p_0 after time t'_0. Hence, ev'' is ordered after ev', although $ev'.vid > ev''.vid$. A contradiction to the assumption that $\mathcal{E}$ is an SVS implementation. ■

5 Weak Virtual Synchrony

In order to define weak virtual synchrony, we introduce a new type of event called *suggested view*. A suggested view event ev includes a view id number, denoted by $ev.vid$, and an ordered list of processes, denoted by $ev.sview$.

A process history h_i is *weakly admissible* if the following holds:

a. If h_i includes a crash event ev, then ev is the last event in h_i.

b. For every send event ev in h_i, there exists a suggested view event ev' such that $ev' \xrightarrow{h_i} ev$, $ev'.vid = ev.vid$, $ev.target \subseteq ev'.view$, and there does not exist a suggested view event ev'' such that $ev' \xrightarrow{h_i} ev'' \xrightarrow{h_i} ev$.

An execution σ is *weakly admissible* if every history in σ is weakly admissible and for every receive event ev in every history h_i in σ, there exists a send event ev' in $h_{ev.pid}$ such that $ev.msg = ev'.msg$ and $p_i \in ev'.target$.

Definition 5.1 (Weak View Admissibility) *An execution σ is weakly view admissible if it is weakly admissible and the following holds:*

a. σ obeys all the conditions in the definition of a strong view admissibility.

b. In every history there exists at least one suggested view event before the first view event and between every two consecutive view events.

c. For every history h_i and every two events ev and ev' such that $ev \xrightarrow{h_i} ev'$ and each of ev and ev' is either a view event or a suggested view event, $ev.vid \leq ev'.vid$. In particular, if ev' is a suggested view event, then $ev.vid < ev'.vid$.

d. For every suggested view event ev and every view or suggested view event ev' such that $ev \xrightarrow{h_i} ev'$ and there does not exist another view event ev'' such that $ev \xrightarrow{h_i} ev'' \xrightarrow{h_i} ev'$, $ev'.view \subseteq ev.view$ and $ev.vid \leq ev'.vid$. Moreover, if a process $p_j \xrightarrow{ev'.view} p_k$, then $p_j \xrightarrow{ev.view} p_k$.

By requirement (d) in Definition 5.1, every view or suggested view, except for the first suggested view after a view event, must be an ordered subset of the previous suggested view. In particular, this means that new processes can only join in the first suggested view that follows a view event; other suggested view events, and the view event itself, can only eliminate processes from previously suggested views.

We now slightly modify the definition of the view event that generates the view in which a message is sent, and introduce the notion of the suggested view which proposes a view to a process. Let h_i be some history, ev a send event in h_i, ev' a view event, and ev'' the first suggested view event such that $ev'' \xrightarrow{h_i} ev'$ and there does not exist another view event ev''' such that $ev'' \xrightarrow{h_i} ev''' \xrightarrow{h_i} ev'$. If $ev''.vid \leq ev.vid \leq ev'.vid$, then we say that ev' *generates the view in which $ev.msg$ is sent*, while ev'' *proposes to p_i the view in which $ev.msg$ is sent.* Note that by this definition, a message can be sent before the event that generates the view in which it is sent.

Definition 5.2 (Weak Virtual Synchrony) *An execution σ is weakly virtually synchronous if the following holds:*

a. σ is weakly view admissible.

b. For every history h_i in σ and every receive event ev in h_i, if ev' is the suggested view event that proposes to $p_{ev.pid}$ the view in which $ev.msg$ is sent, then $ev' \xrightarrow{h_i} ev$. Moreover, if ev'' is the view event that generates the view in which $ev.msg$ is sent and there exists a view event ev''' in h_i such that $ev'''.vid > ev''.vid$, then $ev \xrightarrow{h_i} ev'''$.

c. σ obeys Conditions 3, 4, and 5 in the definition of a strong virtual synchrony.

Note that by requirement (b) in Definition 5.2, the suggested view event that proposes the view in which a message was sent must be seen by all the processes that receive this message. For example, consider histories h_i and h_j that appear in Figure 2. In this example, v_2 proposes to p_i the view in which m is sent, while v_3 generates the view in which m is sent. p_j, which receive m, must see v_2 before the receive event of m. Moreover, the receive event of m can be ordered either before or after v_4, but must be ordered before v_5.

A protocol $\mathcal{P}$ implements weak virtual synchrony if every execution generated by it is weakly virtually synchronous. For the rest of this paper, we use WVS as shorthand for weak virtual synchrony.

The lower bound shown in Lemma 4.1 is valid for weak virtual synchrony as well. However, in this case, there is no need to stop sending messages when the view change protocol is being run. Instead, whenever there is a problem, a suggested view which is a composition of the old view and every process that wishes to join can be introduced to the processes in it. A process that receives a suggested view can send his messages in this suggested view, knowing that some members of the suggested view can be later removed from this view. Finally, when all the messages that were sent in the old view are delivered to all live processes and all the processes that are suspected of being faulty were identified, the "real" new view is introduced to the processes that remains in the "real" view.

6 Implementing WVS

In this section we describe the details of a possible implementation of weak virtual synchrony. Our implementation makes the following assumptions:

1. The underlying environment provides reliable (best effort) FIFO communication. That is, if a process p_i sends a message to another process p_j, then the underlying system will do its best to deliver the message, until either p_j receives that message, or p_i decides to remove p_j from its view. Also, every two messages that were sent by the same process to the same process are delivered in the order they were sent.

2. If a process p_i stops receiving messages of another process p_j, either because p_i really crashes, because the communication links become too lossy, or because p_i has eliminated p_j from it's view, then the failure detector of p_j will eventually generate a suspect event that includes p_i.

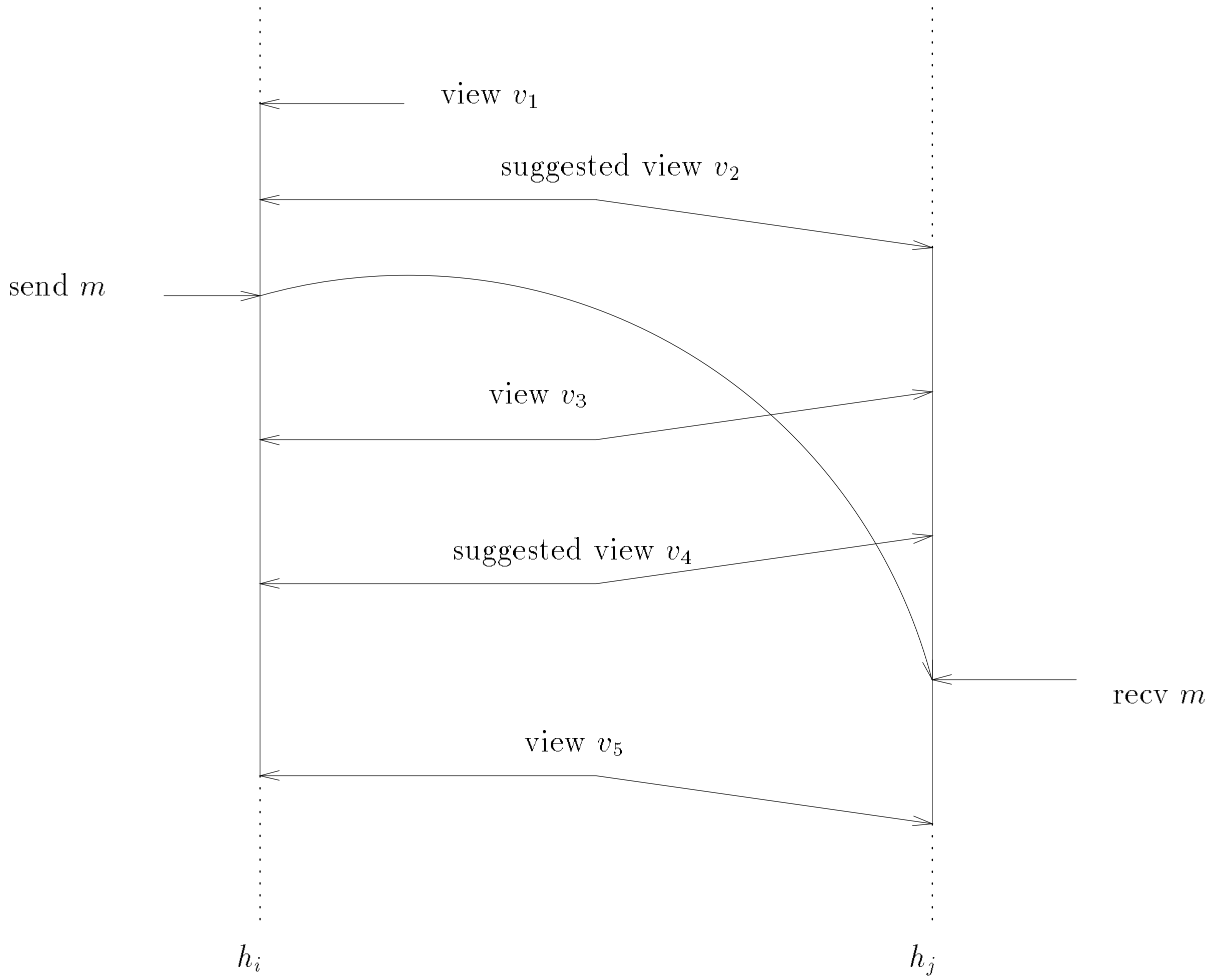

Figure 2: A weakly virtually synchronous execution

3. If a message becomes *stable*, i.e., received by every live member of the view, then every live process in the view will eventually learn about it. (We say that a view is stable if all live processes in that view have received it.)

4. Each message is broadcast to all live processes in the suggested view of its invoking process (at the time the message is invoked).

Assumptions 1 and 2 are supported by the NAK layer of Horus, while Assumption 3 is supported by the STABLE layer of Horus. Assumption 4 is a restriction on the layers which run on top of the WVS layer.

Basically, the algorithm goes as follows: Every process initially constructs a view that consists only of itself, and declares itself the contact for that view. Whenever a contact of a view which is already stable learns about another reachable contact with a smaller address, it sends the smaller contact a join request, which includes its view. Note that if the view is not stable, it is not safe to try to join another group; otherwise, some members of the current view may receive the next suggested view before receiving the current view.

A contact that receives a join request, or suspects that a member of its view has failed, and that is not already busy with a view change, starts a view change. This is done by adding to the current view every process that wishes to join (only if this is the first suggested view after a view), and by deleting all processes that are presumed to be faulty. The initiator of a view change also eliminates from the suggested view processes that wish to join, but appear in the current view. This is done to satisfy Condition 5 in the definition of SVS, which must be satisfied by the definition of WVS too: These processes thought that they were separated from the rest of the view and may have refused to receive some of the messages that were received by the rest of the view. Hence, we must eliminate them before allowing them to rejoin. Similarly, whenever a contact has two pending join requests whose views intersect, the contact ignores one of them.

If a process learns about a new suggested view, it adopts this suggested view, and sends all unstable messages from faulty processes to the initiator of the view change. This process then schedules a **flushed** message to be sent to the new contact immediately after all the messages it has sent in the previous view become stable. Also, if this process was the contact of the previous view, it stops acting as a contact.

If all **flushed** messages arrive, the initiator of the suggested view adopts the suggested view as the new view and sends all unstable messages that it knows of followed by the new view to all other processes in this view. On the other hand, if before receiving all **flushed** messages, the initiator of a suggested view receives an indication that a member of this suggested view has failed, then a new suggested view that does not include this faulty member has to be initiated, as described before.

A non-contact process that receives a new view, adopts it. On the other hand, a non-contact process that thinks that all lower ranked processes in its view are faulty, declares itself a contact and initiates a new view change without the processes that it thinks are faulty.

However, before it sends the newly suggested view to other processes, it must send all unstable messages of the previous contact, so if one of these messages was a view or a suggested view, it will be delivered everywhere before the newly suggested view.

6.1 Pseudocode

Each process maintains its own copy of the following variables:

view_state	Indicates the state at which the view is. The possible values for *view_state* are **running** and **reforming**.
suspects	The list of processes which are suspected of being faulty.
faulty	The list of processes that were declared faulty.
joiners	The list of pending join requests.
suggested_view	The list of processes that appear in the suggested view.
view	The list of processes that appear in the view.
contact	The id of the contact of the view.
contacting	Holds the id of a process it is trying to contact in order to join its group.
svid	The suggested view identifier, consists of a sequence number and process id.
vid	The view identifier, consists of a sequence number and process id.
flush_cnt	Counts how many **flushed** messages were received for the current suggested view.

Note that *svid* and *vid* can be compared lexicographically, to determine if a certain (suggested) view is "older" or "newer" than another (suggested) view.

When sending and receiving messages, processes use a temporary variable *msg*, which is a structure that consists of the following fields: type – **sview**, **flushed**, **join**, **view**, and **cast**; svid – the value of *svid* when it was sent; vid – the value of *vid* when it was sent; contact – the contact of the view; sview – the value of *suggested_view* when it was sent; view – the value of *view* when it was sent; faulty – the value of *faulty* when it was sent. We also make a distinction between the process the *initiates* a message, and a process that *sends* the message, since unstable messages can be retransmitted. Processes are not willing to accept messages that are sent from failed members. However, in order to guarantee virtual synchrony, they accept messages that were originated by a faulty member and retransmitted by a live member.

In the code of the algorithm, we denote by $\cdot$ the concatenation of several lists. That is, $\cdot$ preserves the order among members of the original lists and has the property that if one member of list A appears in the concatenation before a member of list B, then all the members of list A will appear in the concatenation before all members of list B.

Although most of the code is given as in an event driven form, there are two procedures, **init_view_change** and **create_new_view**. These procedures are presented in Figure 3: **init_view_change** is responsible for generating a new suggested view and distributing it among the members of that suggested view; **create_new_view** is responsible for adopting a suggested view as the next (real) view, and distributing this information among the members of this view. The code for handling net_receive events (messages that are received from the network) is given in Figure 4. In Figure 4 we assume that p_i is the process that sent the message. Finally, the code for handling other events is given in Figure 5.

6.2 Performance Measurements

In this section we compare the latency of messages sent during view change using both the SVS protocol that is used by Horus and the WVS protocol described in this paper, under various background loads. These measurements were taken on 4 Sparc-20s, connected by a 10 Mbps Ethernet. In order to create the background load, each machine broadcast a message to all other machines in fixed intervals of time. The load on the system is increased by shortening this interval, and is decreased by lengthening this interval. The latency measurements were taken by timestamping each message at each of the layers in Horus, on its way to the Ethernet, and, by using a special device driver, immediately before going out to the Ethernet. At the receiver side, messages were timestamped by our device driver immediately when they were received from the Ethernet, and then at each layer. Using these timestamps, we were able to collect accurate measurements of the one-way latency from the application to the network, denoted by δ_1, and the one-way latency from the network to the application, denoted by δ_2. If we assume that the time that messages of the same size spend on the wire is the same, then by adding δ_1 and δ_2, we get a good estimate of the *behavior* of the real latency.

Since we are running on real systems, there is always the problem of operating system's interference with the measurements. Luckily, all the measurements that we got were either within a factor of three of the average, or at least two order of magnitudes higher than the average. This allowed us to identify measurements which were probably the result of operating systems' related issues, e.g., context switches, and to discard these measurements from the latency graphs.

The results of our measurements appear in Figure 6. As can be seen, the latency of messages sent during view changes in the WVS implementation is higher than the latency of messages sent during normal operation by only a small constant which does not depend on the background load. We explain this difference by the extra overhead associated with handling a suggested view event. On the other hand, messages sent during view changes in the SVS implementation were significantly slower than those sent in the WVS implementation, and the gap between the two implementations increases with the background load.

Also, the results reported in Figure 6 were obtained under fairly simple failure modes, that could have been generated in a systematic manner. Since under more complex failure scenarios it may take much longer to recalculate the new view, we anticipate that in these failure modes the difference between SVS and WVS is even more significant.

```
procedure init_view_change
    faulty := faulty ∪ suspects
    suspects := nil
    if view_state = running then
        suggested_view := suggested_view · {ev.view | ev ∈ joiners}
    endif
    if contact ≠ me then
        rebroadcast all unstable messages of contact to every member
                                            in suggested_view \ faulty
    endif
    contact := me
    msg.type := sview
    msg.sview := suggested_view
    msg.contact := contact
    msg.faulty := faulty
    msg.svid := svid := ⟨ max(svid, {ev.svid | ev ∈ joiners}) + 1, me ⟩
    generate a suggested view event with suggested_view and faulty
    broadcast msg to every process in suggested_view \ faulty
    flush_cnt := 0
    view_state := reforming
endproc

procedure create_new_view
    view := suggested_view := suggested_view \ faulty
    oldvid := vid ; vid := svid
    faulty := nil
    msg.type := view
    msg.view := view
    msg.vid := vid
    view_state := running
    generate a view event with view
    rebroadcast all unstable messages from previous view that I know
                                            of to every process in view
    broadcast msg to every process in view
    if joiners is not empty then call init_view_change endif
endproc
```

Figure 3: The WVS protocol – procedures

```
if msg.type = sview and msg.svid > svid and contacting = ⊥ then
    if vid ≠ svid or msg.sview and msg.faulty do not form an ordered subset of
                suggested_view and faulty then
        quit
    endif
    send every unstable message that was originated by a faulty process to p_i
    view_state := reforming
    suggested_view := msg.sview
    svid := msg.svid ; contact := msg.contact
    faulty := msg.faulty ; suspects := suspects \ faulty
    msg.type : = flushed ; msg.svid : = svid
    schedule to send msg to p_i immediately after all my messages send in the previous
                                                    view become stable
    generate a suggested view event with suggested_view and faulty
elseif msg.type = flushed and msg.svid = svid and view_state = reforming then
    flush_cnt := flush_cnt + 1
    if every process in the suggested view returned a flushed message then
        call create_new_view
    endif
elseif msg.type = join and contact and contacting = ⊥ and msg.view does not overlap
                with the view of other join requests in joiners then
    if msg.view ∩ suggested_view ≠ ∅ then
        add every process in msg.view ∩ suggested_view to suspects
        call init_view_change and quit
    endif
    ev.view := msg.view ; ev.svid := msg.svid
    ev.pid := p_i
    joiners := joiners ∪ ev
    if view_state = running then call init_view_change endif
elseif msg.type = view and msg.vid = svid and msg.view = suggested_view \ faulty then
    view := suggested_view := msg.view
    oldvid := vid ; vid := msg.vid
    faulty := suspects := nil
    generate a view event with view
elseif msg.type = cast and msg.svid > oldvid and p_i ∈ suggested_view and
                                                    p_i ∉ faulty ∪ suspects then
    generate a receive event with msg
endif
```

Figure 4: The WVS protocol – handling messages (sent by p_i)

```
Whenever the failure detector indicate that process p_i is faulty do:
    if contact then
        if contacting = p_i then contacting := ⊥
        else
            add p_i to suspects
            if contacting = ⊥ then call init_view_change endif
        endif
    elseif all process that appear before me in suggested_view seem to be faulty then
        add them to suspects and call init_view_change
    else
        add p_i to suspects and inform the failure detector of the lowest ranked
                                   live process in suggested_view about p_i
    endif
enddo

if contact and contacting = ⊥ and view_state = running and
        the last view is stable and p_i is a smaller but reachable contact then
    contacting := p_i
    msg.type := join
    msg.view := view
    send msg to p_i
endif

Whenever the application generates a send event with message m do:
    msg.svid := svid
    msg.msg := m.msg
    msg.type := cast
    broadcast msg to every process in suggested_view \ faulty
enddo
```

Figure 5: The WVS protocol – handling local events

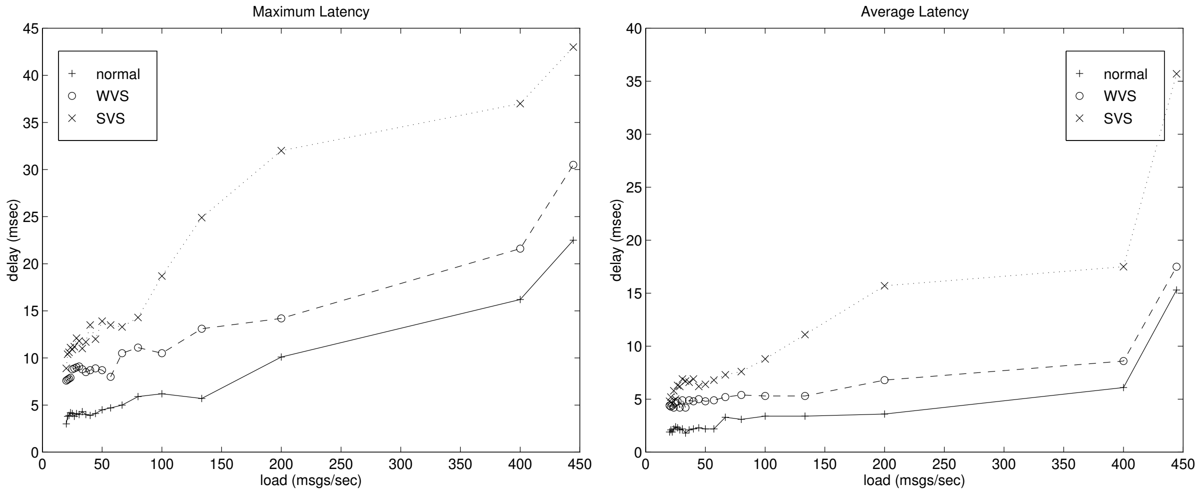

Figure 6: Latency measurements

6.3 Implementing SVS on top of WVS

The following lemma states that if the application program would stop sending messages between suggested view events and the following view events, then the resulting semantics would be of strong virtual synchrony.

Lemma 6.1 *Let σ be a WVS execution in which for every history h_i and every send event ev in h_i, ev is ordered in h_i after the view event that generates the view in which ev.msg is sent. Then σ is an SVS execution.*

If the WVS layer is implemented using the protocol described in Section 6, then the following scheme can be used to optimize the performance when a process experiences several suggested views before a "real" view is installed: Whenever the application receives a suggested view event, it sends all the messages awaiting to be sent in the "old" view, and only then it allows the membership layer to issue the flush message. This way, messages may never have to be delayed for more than one round of flush messages, although the view change itself may take up to $t + 1$ rounds, if t processes crash during this view change.

7 Ordered Virtual Synchrony

In this section we define ordered virtual synchrony and discuss how Horus' implementation of ordered virtual synchrony can benefit from using WVS instead of SVS.

In order to define ordered virtual synchrony, we must introduce the notion of *causal order*, originally defined by Lamport in [15]. Given an execution σ, we say that event *ev causally* *precedes* event ev' if (a) $ev \xrightarrow{h_i} ev'$ for some history h_i, (b) ev is a send event, ev' is a receive event and $ev.msg = ev'.msg$, or (c) there exists another event ev'' such that ev causally precedes ev'' and ev'' causally precedes ev'.

We assume that send events can be labeled as either *abcast*, *cbcast*, or *unordered*. Given an execution σ, we define a partial order $\xrightarrow{\sigma}$ on the receive and send events of σ as follows: Let ev and ev' be two receive events in σ and let ev'' and ev''' be the corresponding send events, respectively. Then the following holds:

1. If ev'' and ev''' are labeled as abcast, then either $ev \xrightarrow{\sigma} ev'$ or $ev' \xrightarrow{\sigma} ev$. Moreover, if $ev'' \xrightarrow{h_i} ev'''$ for some i, then $ev \xrightarrow{\sigma} ev'$.
2. If ev'' is labeled as cbcast, then ev is ordered in $\xrightarrow{\sigma}$ after any event the causally precedes it in σ, and before any event that causally follows it in σ.

We denote by $\xrightarrow{\sigma} \mid i$ the restriction of $\xrightarrow{\sigma}$ to events that occur in h_i. Given a sequence of events h and a partial order $\xrightarrow{\sigma}$ on the set of events in h, we say that h *extends* $\xrightarrow{\sigma}$ if for every two events ev and ev' in h such that $ev \xrightarrow{\sigma} ev'$, $ev \xrightarrow{h} ev'$.

Definition 7.1 (Ordered Virtual Synchrony) *An SVS (or WVS) execution σ is strongly (or weakly) ordered virtually synchronous if there exists a partial order $\xrightarrow{\sigma}$ such that for every process p_i, h_i extends $\xrightarrow{\sigma} \mid i$.*

Horus implements ordered virtual synchrony in two layers, called *total* and *causal*. We now discuss how these layers of Horus can benefit by having the lower layers implementing WVS instead of SVS.

7.1 Total Delivery and Causal Delivery

It is clear that total delivery and causal delivery can be implemented with weaker guarantees than those provided by SVS. However, it is also clear that the total layer cannot deliver a message sent in a new view until all messages of the old view are guaranteed to be delivered. This is due to the possibility that some messages that are missing from the old view were delivered in some of the processes before any of the messages of the new view. However, we believe that a protocol for implementing total delivery can still benefit from WVS since even in cases as described above, messages are delayed at the receiver side and not at the sender side. Hence, when the "real view" is installed, the delivery of these messages does not require any network traffic.

As for the causal layer, in the worst case, a message may have to wait for the new view to be installed, like in the total layer, before it can be delivered. However, from our experience, messages of a new view can often be delivered before the "real" view is installed, since usually none of the missing messages from previous views causally precede them. In particular, local deliveries do not have to be blocked at all.

8 Multiple Groups

Up until now, the discussion about virtual synchrony was limited to the case were there is only one group. However, in practice, there can be many groups and every process may be a member of several groups simultaneously. This, of course, can be modeled in our framework by adding a new field called *group* to every event and then use the definitions of Sections 4 and 5 on the restriction of executions and/or histories to events of the same group. For example, an execution σ is SVS (WVS) if for every group g, $\sigma \mid g$, the restriction of σ to events of group g, is an SVS (WVS) execution according to the definitions of Section 4 (5).

Although the above definition correctly describes the guarantees provided by Horus, Horus provides an additional guarantee known as *failure domains*. That is, if a process is a member of several groups and is declared or suspected of being faulty, then it is declared or suspected of being faulty in all the groups it belongs to. The following is a suggested definition for failure domains:

Definition 8.1 (Failure Domain) *Given a set of groups G, we say that G shares the same* failure domain *if the following holds: For every history h_j and every process p_i, if h_j includes a suspect event ev such that $p_i \in ev.suspects$, then for every group $g \in G$ for which h_j includes a view event ev' such that $ev'.group = g$ and $p_i \in ev'.view$, there exists another event ev'' such that $ev' \xrightarrow{h_j} ev''$, $ev''.group = g$, and either ev'' is a suspect event such that $p_i \in ev''.suspects$ or ev'' is a view event such that $p_i \notin ev''.view$.*

This definition is slightly weaker than the behavior of failure domains when a process is suspected of being faulty more than once. Unfortunately, we are currently not aware of any simple way to define this behavior formally and precisely.

9 Discussion

Our current research with Horus is aimed at developing a tool that will provide the application with a semantics which is strong enough to cope with failures, while ensuring low message latencies, even in "bad" scenarios. In this paper we take a step towards this goal, by looking at the membership layer of Horus. We presented a formal definition of strong virtual synchrony, capturing the guarantees provided by this layer. Using this formal definition, we were able to identify sources for long delays in this layer, and to suggest a way to overcome these delays without sacrificing the desired properties of the existing semantics, namely, by introducing the notion of weak virtual synchrony. We have developed an implementation for weak virtual synchrony and showed that indeed, in our system, weak virtual synchrony does not incur the same long latencies as strong virtual synchrony.

In addition to the membership layer, Horus has several other layers that may incur unbounded latencies. Hence, we would like to study these layers too, and make sure that, at least with very high probability, we can eliminate the sources of uncertainty from these layers as well. At the moment we are also studying the layer that is responsible for providing total delivery of messages, and are looking at the behavior of several total delivery protocols.

In order to reach a higher degree of confidence in our protocols, we would like to formally prove their correctness. This can be either by composing formal proofs of correctness, or by using automatic verification tools. For this, the formal definitions presented in this papers will be very helpful.

We believe that it is possible to slightly modify the protocol developed in Section 6 to make it self stabilizing [11] under certain assumptions. For example, if we assume that processes have a unique name, that cannot be corrupted, then we believe that by adding several sanity checks, this protocol would become self stabilizing. If the processes' name can be corrupted, but they have a way to retrieve it, e.g, the name of a process is the address of the network device and this address cannot be corrupted, then we believe that by verifying the name every so often, and by adding some sanity checks, the protocol would also become self stabilizing.

Acknowledgements: We would like to thank Ken Birman, David Karr, and Werner Vogels for many helpful discussions and comments. We would also like to thank Gil Neiger and Özalp Babaoğlu.

References

[1] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Transis: A Communication Sub-System for High Availability. In *Proc. of the 22nd Annual International Symposium on Fault-Tolerant Computing*, pages 76–84, July 1992.

[2] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Membership Algorithms in Broadcast Domains. In *Proc. of the 6th International Workshop on Distributed Algorithms, Lecture Note in Computer Science #647*, pages 292–312, November 92.

[3] Y. Amir, L. E. Moser, P. M. Melliar-Smith, D.A. Agarwal, and P. Ciarfella. Fast Message Ordering and Membership Using a Logical Token-Passing Ring. In *Proc. of the 13th International Conference on Distributed Computing Systems*, pages 551–560, May 1993.

[4] Ö. Babaoğlu, R. Davoli, L. Giachini, and M. Baker. Relacs: A Communication Infrastructure for Constructing Reliable Applications in Large-Scale Distributed Systems. Technical Report UBLCS–94–15, Department of Computer Science, University of Bologna, June 1994. Revised January 1995.

[5] K. Birman. The Process Group Approach to Reliable Distributed Computing. *Communications of the ACM*, 36(12):37–53, December 1993.

[6] K. Birman. Integrating Runtime Consistency Models for Distributed Systems. *Journal of Parallel and Distributed Computing*, 23:158–176, 1994.

[7] K. Birman and T. Joseph. Exploiting Virtual Synchrony in Distributed Systems. In *Proc. of the 11th ACM Symp. on Operating Systems Principles*, pages 123–138, December 1987.

[8] K. Birman and T. Joseph. Reliable Communication in the Presence of Failures. *ACM Transactions on Computer Systems*, 5(1):47–76, February 1987.

[9] K. Birman, A. Schiper, and P. Stephenson. Lightweight Causal and Atomic Group Multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, August 1991.

[10] J. Chang and N. Maxemchuk. Reliable Broadcast Protocols. *ACM Transactions on Computer Systems*, 2(3):251–273, August 1984.

[11] E. W. Dijkstra. Self-stabilizing Systems in Spite of Distributed Control. *Communications of the ACM*, 17(11):643–644, November 1974.

[12] P. Ezhilchelvan, R. Macedo, and S. Shrivastava. Newtop: A Fault-Tolerant Group Communication Protocol. Technical report, Computer Science Department, University of Newcastle, Newcastle upon Tyne, United Kingdom, August 1994.

[13] A. Gopal and S. Toueg. Reliable Broadcast in Synchronous and Asynchronous Environments. In J.-C. Bermond and M. Raynal, editors, *Proceeding of the Third International Workshop on Distributed Algorithms*, volume 392 of *Lecture Notes on Computer Science*, pages 110–123. Springer-Verlag, September 1989.

[14] R. Guerraoui. Transaction model vs Virtual Synchrony model: bridging the gap. In K. Birman, F. Cristian, F. Mattern, and A. Schiper, editors, *Distributed Systems: From Theory to Practice*, Lecture Notes on Computer Science. Springer-Verlag, 1995. To appear.

[15] L. Lamport. How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE Trans. on Computers*, C-28(9):690–691, 1979.

[16] B. Liskov, S. Ghemawat, R. Gruber, P. Johnson, L. Shrira, and M. Williams. Replication in the Harp File System. In *Proceeding of the 13th ACM Symposium on Operating Systems Principles*, pages 226–238, Pacific Grove, CA, October 1991.

[17] S. Maffeis, W. Bischofberger, and K. Matzel. A Generic Multicast Transport Service to Support Disconnected Operation. In *Proceeding of the 2nd USENIX Symposium on Mobile and Location-Independent Computing*, pages 79–89, Ann Arbor, MI, April 1995.

[18] M. Makpangou and K. Birman. Designing Application Software in Wide Area Network Settings. Technical Report 90-1165, Departmenr of Computer Science, Cornell University, October 1990.

[19] S. Mishra, L. Peterson, and R. Schlichting. Consul: A Communication Substrate for Fault-Tolerant Distributed Programs. *Distributed Systems Engineering Journal*, 1(2):87–103, December 1993.

[20] G. Neiger and S. Toueg. Automatically Increasing the Fault-Tolerance of Distributed Algorithms. *Journal of Algorithms*, 11(3):374–419, September 1990.

[21] L. Peterson, N. Buchholz, and R. D. Schlichting. Preserving and Using Context Information in Interprocess Communication. *ACM Transactions on Computer Systems*, 7(3):217–246, August 1989.

[22] A. Ricciardi and K. P. Birman. Consistent Process Membership in Asynchronous Environments. In K. Birman and R. van Renesse, editors, *Reliable Distrinuted Computing With The ISIS Toolkit*, chapter 13. IEEE Computer Society Press, Los Alamitos, 1993.

[23] A. Schiper and A. Sandoz. Uniform Reliable Multicast in a Virtually Synchronous Environent. In *Proc. of the 13th International Conference on Distributed Computing Systems*, pages 561–568, May 1993.

[24] R. van Renesse, K. Birman, R. Friedman, M. Hayden, and D. Karr. A Framework for Protocol Composition in Horus. In *Proc. of the 14th ACM Symposium on Principles of Distributed Computing*, pages 80–89, August 1995.

[25] R. van Renesse, K. P. Birman, and T. M. Hickey. Design and Performance of Horus: A Lightweight Group Communications System. Technical Report 94-1442, Cornell University, Dept. of Computer Science, August 1994.

[26] R. van Renesse, R. Friedman, and D. Malki. Understanding Virtual Synchrony. In preparation.