Conditions for the State Transfer on Virtual Synchronous Systems

Luis M. Peña Cabañas, Juan Pavón Mestras

Departamento de Sistemas Informáticos y Programación, Universidad Complutense de Madrid

Ciudad Universitaria s/n, 28040 Madrid, e-mail: luismp@skynet.be, jpavon@sip.ucm.es

Abstract*: As stated in the virtual synchrony model, members joining dynamic groups can obtain the group's current state from some prior member or from a set of members. Existing group communication systems drive that state transfer in different ways, usually blocking some or all the group members during the state transfer. This paper formalizes the state transfer problem, formulating the generic protocols needed among the members of a group, even for the transfer of big states that would require one or more messages.*

1. Introduction

A common technique for building reliable distributed systems is the replication of software components on multiple processors. This helps the distributed system to guarantee a quality of service in spite of failures in the underlying computer platform. Still there is a need to coordinate the replicated components so that they appear to the client as a single logical entity. This coordination means that all the active replicas are in a consistent state. The *virtual synchrony* model [Birman87] can be applied to guarantee such consistent state among these replicas. This model works under a *group* paradigm: replicas are defined as members of a group, which appears to clients as a single entity. Group members can be defined as active or passive; only the active members process client requests, and the passive ones just synchronize their states periodically.

Active group members achieve consistency through reliable group communications: an action on any member is propagated to the other members using protocols that provide a reliable message delivery, with a partial (e.g., *FIFO*, *causal*) or *total order* [Lamport78]. Groups are not static in the sense that they can incorporate or remove members dynamically. The list of non-faulty members in a group, which is called a *view*, is maintained by an internal service called *Group Membership Service* (GMS). GMS uses unreliable fault-detectors [Chandra91], commonly based on replicas pinging, to detect and expulse faulty members. When this happens, as well as when a member joins or leaves the group, the *view* changes, and GMS is responsible to install the new view in all members.

Starting from an initial *view*, each admitted group member could interact with the other members. The simplified abstraction seen by developers of groups under the *virtual synchrony* model is that all the members of the group see the same messages in the same order. Because every member sees the same input, they can keep consistent states, if they have started from the same initial point [Schneider90]. This condition implies the need to make a state transfer to the joining members, so that they also start from the same shared state.

The *extended virtual synchrony* model [Babaoglu95] is a variation on the previous model when, because of network partitions, several operational subgroups of the original group may coexist. Under the normal *virtual synchrony* model, only one subgroup can stay operational, and, for this reason, those systems are called *primary groups* [Ricciardi93]. The extended virtual synchrony model considers a future merging of the partitioned groups, what implies the need to reconcile their states. This problem can be simplified if it is handled as a state transfer, by discarding one of the group states, but it is in general a more difficult problem than the state transfer is.

This discussion shows that the virtual synchrony model relies on efficient and reliable state transfer mechanisms, which can be implemented on top of group communication primitives. State transfer mechanisms can also be applied to maintain consistency in web servers, distributed databases, and any system that requires handling of replicated data. As an initial approach, one or more members send their state in a message to the new members. However, this is not generally enough, as the member sending the state and the member receiving it could be target of other group messages. If the new member processes some of those messages before receiving the state, and the member sending it is going to process those messages after sending the state, both members will likely finish in different, inconsistent states. Additionally, the state could be big enough to need several messages to do the transfer, and a new view could be installed between those messages, being also possible that the member sending the state falls down before sending the whole state. Although a group could handle all these possibilities by defining its own protocols, this is a general problem for any group, so it seems desirable to build these protocols inside the group communication logic.

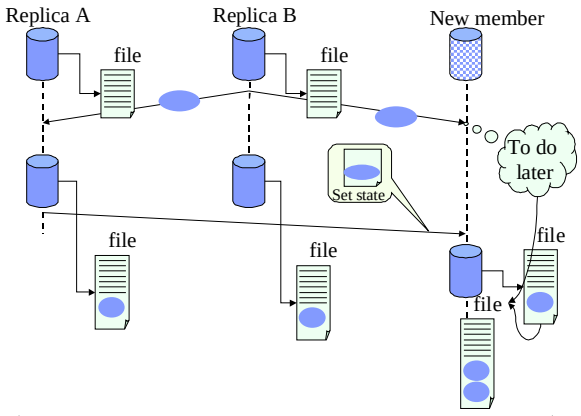

Figure 1

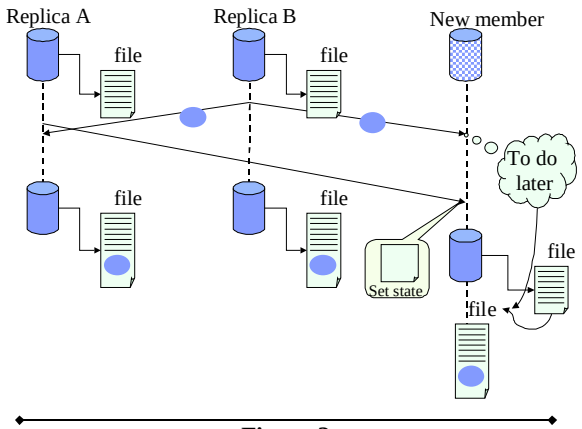

Figure 2

Group communications have been a hot research topic during the last years, and several systems have been built supporting the *virtual synchrony* model. This paper focuses on the problems that could happen on a state transfer, and formulates a generic approach to solve those problems and discuss several variations. Next section defines the state transfer problem and the approach taken by some well-known group communication systems to solve it. We have studied the conditions on groups and messages to allow a simple state transfer (without any special processing) and defined a set of generic protocols that can be applied by any group. These protocols, shown in the third section, are compatible with those used under other group communication systems, and they match them -at least partially- in most of the cases.

In summary, this work formalizes the state transfer problem, formulating the generic protocols needed among the members of a group, even for the transfer of big states that would require one or more messages.

2. The state transfer problem

The goal of the state transfer problem is to achieve a consistent state in all the members in the group when one or more members with not initialized state join an already consistent group. The initial group could have members without state only if a transfer were already being done to at least one of them when the new member was included.

This definition of the problem already excludes one solution: when the future member contacts an existing group member to receive the state and, once received joins the group. This solution would require that the contacted group member would keep track of any changes on the state from the moment where the state was sent until the new member is inserted in the group. Moreover, those changes will have to be transferred anyway, becoming the transfer state problem as it has been specified!

Although a state transfer can be seen as a bi-directional communication between two non empty sets of members in the same group, we first delimitate this problem to the unidirectional communication needed to initialize the state of a joining member with the state shared by the already initialized, consistent replicas. The problem domain is a group working under the *virtual synchrony* model, where some new members join the group (this assumes that there is at least one member with a defined state). The solution must use the group communication primitives supported by the specific *virtual synchronous* model, mainly the reliable multicast messages.

To understand this problem, consider as an example a distributed file system, where each replica contains a copy of a part of the whole file system. The file system consists of several sections, each one replicated on several machines to provide fault tolerance and load balance. The file service is provided by several server groups, each one handling one of its sections, and following the *virtual synchrony* model; messages are reliable and have total ordering (every member sees the same messages in the same order). When a replica becomes active and joins the group, it receives the appropriated part of the file system from another replica in one or more messages; let's suppose also that a third replica sends a message to append a chunk of bytes to a specified file that is in the part of the file system being transferred.

Figure 1 shows one hypothetical scenario, where the replica sending the state receives and processes the new message before sending the state. If the new member receives first the *append* message, it saves it for processing after receiving the full state. Then, it adds the new chunk, terminating in an inconsistent state. However, the *append* message needs to be saved, as shown in figure 2. In this figure, the state is sent before processing the *append* message, and therefore the joining

member must have saved the message to get a final consistent state. There are other scenarios where the message is not saved, but in these cases, there can be obtained no consistency either.

The easiest solution would be to block the whole system during a transfer, but this would decrease system performance dramatically if the state to send is large or there are often view changes. It is clear on the previous problem that it would be enough to buffer every received message in the replica sending the state while this state is not sent to the joining member: if the member sending the state does not process any message before sending the state, and the new member also queues the input messages until the state is received, both members will finish in consistent states. The problem becomes even more complex if the state must be sent in several pieces (for instance, it is not feasible to send the file system in just one message) and the replica sending these pieces falls down before finishing. There is still a generic solution to these problems, and this is precisely the focus of this paper.

It is possible to do the state transfer on any group without special support, just building the protocol into the group logic, but most of the group communication systems known by us give some support to the applications. *Arjuna* [Arjuna, Parrington95] does automatically a state transfer when a new member joins a group. The state transfer is done in one step, blocking the whole system, and group members must just implement two state operations: *save_state, restore_state*. The same approach uses *Electra* [Electra, Maffeis97], but with a more flexible scheme, allowing transfers in several pieces and still using a simple interface: *get_state, save_state*. In *Phoenix* [Phoenix, Malloth96], only the members involved in the transfer are blocked, but the transfer must be done in just one step, with a similar interface: *get_state, put_state*. *JavaGroups* [JavaGroups] does not invoke automatically the transfer; this is an operation to be called by the interested member, which can request the state from every group member or just from one, generally the oldest one. The member sending its state must first make a copy of its state, and while this copy is performed, the member is blocked. When the state is requested, the member just sends this temporal copy. It is a transfer in one step, and requires total ordering on the messages; the interface is slightly more complex, to do the state copy: *SaveState, GetState, and SetState.*

Maestro [Maestro], a *C++* interface for *Ensemble* [Ensemble, Hayden98] allows several approaches to the state transfer. A group can specify if it needs a transfer and, if so, it will be invoked automatically. Messages can be classified as safe or unsafe; in the former case, they can be sent during the transfer, otherwise they are blocked, but this means that the relative order between messages is lost. Additionally, the messages are blocked when they are sent; therefore, if there are messages coming from the previous view (those sent once the view has been blocked to install a new view), they would pass the blocking even when considered unsafe. Transfers can be done in several steps, as opposed to the latest versions of Maestro, which include a new state transfer mechanism in one step. State members automatically send their state, being possible to identify the older one. Therefore, any member can update its state using the newest state. The protocol is simple, and has an acceptable performance when the state is small, groups have few members, and view changes are unusual.

The transfer from another active replica is not the only way; for example, *Cactus* [Cactus, Schlichting93] works under a *checkpoints* scheme. Each replica must store periodically checkpoints of its state, and keep a trace of the messages processed from the last checkpoint. If a replica becomes active, it can build its state from those checkpoints. This technique is fast when replicas are down during short periods. It also requires every replica to spend some processing time periodically, not only in case of a state transfer.

The proposal made to achieve fault tolerance in CORBA through entity redundancy [OMG98] allows the definition of active and passive groups. On both cases, the application can choose to handle its own consistency through its own state transfer mechanisms, or to shift this responsibility to the CORBA infrastructure. In the latest case, objects must implement an interface that defines the operations *get_state* and *send_state*. Optionally, objects can implement a specialized interface, which also defines the operations *get_update, set_update*, to store and retrieve the state incrementally. Every message is stored in a log, which can be distributed, and when a member must receive the state, this can be built from the log. It is first received the state, optionally some updates to this state and finally those messages received after having logged the last state. If the group is defined as *cold passive*, there is just one primary member, which stores its state every 100 nanoseconds (configurable timer). Backup members will only receive the state if the primary member falls down. If defined as *warm passive*, backup members receive the state after each operation. Therefore, after a failure on the primary member, the activation of a backup member takes less time than in the case of *cold passive* groups. However, this approach is only effective when the transfer of the state is less expensive than the processing of a message. Finally, for active groups, the state to be transferred is built from the *log*, and no member must be blocked during the transfer (except the new one). On the other side, as with the *Cactus* system, replicas are spending time continuously storing their states.

Following sections study the conditions to be satisfied by the groups and their messages on state transfers from one or more active replicas. The blocking of the involved replicas is generally needed, except in the simplest cases, but the

blocking and filtering of messages is shown to be dependant on the messages ordering. The state transfer in several steps is also driven, a requirement to be considered when the group state is large enough.

3. Model and definitions

A conclusion of this paper states that even if there is no blocking on any member during a state transfer, it is possible to build the state from a set of messages and a message that contains the state that has been reached after having processed a subset of those messages. It must be taken into consideration that the order on which those messages and the state message is received depend on the application ordering restrictions, and, except when using total ordering, this order cannot be predicted at all. Although this conclusion may seem quite natural, we will formulate the problem and demonstrate a solution from a formal perspective. This requires first introducing a model for a distributed system with replicated software components, as well as the corresponding basic notation.

Each member has a state, which consists of a private and a common part, the second to be shared with the group. The state transfer problem is concerned with the common part. Two members have consistent states if, given that no new messages are sent in the group and no external interaction is performed, the processing of every pending message drives them to the same final state. The *state message* is the message sent from a state member that is called *transfer coordinator* or simply *coordinator* to a joining member, and includes the state of the member in that moment. There can be several state messages if the state is split in several chunks.

The distributed system consists of replicated software components in groups, and a GMS that follows these rules:

1. Groups are built using the virtual synchrony model. Weak virtual synchrony [Friedman95] is not taken in account in this paper, for simplicity, a further discussion is given in [Pena99tr].
2. State changes in one member due to some external interaction must be propagated to the rest of members in the group, using the group communication primitives. Two behaviors are allowed: a member changes its state and communicates it to the group (behavior dynamically non-uniform) or it does first the communication and only when the rest of members have received the messages, it changes its own state (dynamically uniform). In the second case, we consider that the member itself receives its own message to process it at the same time as the rest of members in the group.
3. Members behave deterministically: two members with the same state receiving the same messages in the order specified by the application will evolve to the same final state.
4. Joining members do not take any action before the first view is installed and it is accepted in the group.

The interaction between the *GMS* and the member is modeled as follows:

1. Members receive messages sequentially from other members and from the GMS: one message each time, not receiving the next one after the previous has been processed. Note that a buffering of the message in the member (the member queues internally to be processed afterwards) will therefore be considered as having been processed, as the GMS can not be aware of that buffering. Depending on the application, messages from other members can be handled concurrently, but the messages sent from the GMS are sequential with respect to other messages from the GMS or from other members.
2. The GMS installs a new view only when all the messages in the previous view have been processed. The GMS is considered to be blocked until the view is installed; if the member is buffering messages, the view is only installed after they have been extracted from the buffer and processed.
3. The GMS reports a blocking condition to the members (when a view is blocked, any new message sent would only be sent to the group in the next view).
4. During the blocked period, a member can still send messages, but they are queued in the GMS. After the view is installed, they are sent to the other members. The messages sent by one member can be ordered as *FIFO* (if required by the application), but messages sent by different members are considered to be concurrent.

3.1. Notation

The previous discussion can be expressed more formally with the following notation.

State: $S=s+s'$. The member's state is split in a global part (s) and a private one (s'). The state transfer problem is concerned only with the global, common part.

Group members: $G=\{g_1, g_2 \dots g_n\}$ $(n \geq 1)$. The group includes every member, with or without state. GS={g1, g2 … g_m} / (n ≥ m) represents the list of initialised members (sharing a common state), and can be empty. The not initialized members are included in GU, which can be also empty. This notation assumes that members are directly included in the group, and the transfer is only started after the members join the group.

Succession of states: $s_{iv}=\{s_{iv1}, s_{iv2} \dots s_{ivl}\}$ Each member *i* has on each view *v* a succession of states. When the view information is not useful, it is removed from the notation: $s_i=\{s_{i1}, s_{i2} \dots s_{il}\}$. On a strongly synchronous system, that is, with strong virtual synchrony and dynamically uniform messages with total causal order, every initialized member proceeds the same sequence of states: $s_{in}= s_{jn}, \forall i,j \leq m$.

Messages

1. M_{ij} is the ordered set of messages already processed by j but not by i, which have been sent or must be sent to i (if j does not fall down before)
2. $M_i = \sum_{j \neq i} M_{ij}$: Messages to be processed by i because other members have already processed them.
3. $M_{ij} = \phi,\ j \geq m$: joining members do not take any action before being accepted in the group.
4. If a group is designed to take actions only after receiving a message, no members can have processed any message that any other member had not processed when a new view is installed. These members react to external events by sending a message to the others, and taking the action only when that message is received. Therefore: $M_{ij} = \phi,\ \forall i,j$. We name these messages *loopback messages*; messages processed with total ordering or dynamically uniformly are *loopback* messages.
5. $M_{ij} = \phi,\ \forall i,j \leq m \Rightarrow s_{iv1}= s_{jv1}, \forall i,j \leq m$.: If M_{ij} is null for any pair of members, every state member has the same state after a view is installed.
6. M_{np} Is the ordered set of messages already sent by any group member but not yet processed by any of them. If messages are not dynamically uniform and are sent by a member after having processed it, then $M_{np} = \phi$. Groups with total ordering or messages dynamically uniform will have non empty M_{np}. If the order is other than total, it will be empty except when members react exclusively to the reception of messages (*loopback* messages, the sender of a message must also process it).
7. Messages ordering**:** $M \oplus M'$ means to process both groups of messages in the order specified by the application, while $M + M'$ means to process first *M* and later *M'*.
8. Set of messages. $M = M'$ means that both sets of messages are exactly the same, including the messages order. $M \approx M'$ means that both sets include the same messages, but the order can be different, depending on the application restrictions.
9. Messages subset: $\overline{M}$ means a subset of the messages in M **.**

State message. $ms / s_O + ms = s$: A member on its initial state receiving a state message adopts the state included in the message. If the state is sent in several chunks, the previous rule is specified as: $s_O + (ms_1 \oplus ms_2 \oplus \dots \oplus ms_n) = s$. *In* general, ms_i^j means the j[th] chunk of state s_i, and M^j is the group of messages in *M* that modify the j[th] chunk; note that these same messages could modify other chunks as well.

4. State transfer requirements

For the purposes of this presentation, we impose several restrictions that are later removed: no view changes are produced during the state transfer, the state is sent in just one message, and no messages are sent inside the group during the transfer. We consider the entire group of messages, included the state message to be restricted to the same order. These requirements overlook the mechanisms to detect when a transfer is finished or to detect when there are no members with state.

Let's suppose three members, one of them to receive the state. The members with state, g1 and $g_{2,}$ may start with a different initial state, but will reach the same final state after processing the pending messages. These messages are those sent in the previous view during the blocked period, and are different for each member:

$$g_1 : Mnp \oplus M_1 \quad ,being \quad M_1 = M_{12} \quad g_2 : Mnp \oplus M_2 \quad ,being \quad M_2 = M_{21}$$

Any member with state can be the coordinator; they will process a subset of the pending messages to get an intermediate state (they do not have to be the same on both members), and will send the state message in that moment; afterwards, they will process the rest of messages. If we split the set of messages to process in two subsets, it is possible to write:

$$g_{1:}\begin{bmatrix} messages: & Mnp \oplus M_{12} = M_A + M_B \\ state: & \begin{bmatrix} s_{I1} + M_A = s_1 & (state \quad sent) \\ s_1 + M_B = s_F \end{bmatrix} \end{bmatrix} \quad g_{2:}\begin{bmatrix} messages: & Mnp \oplus M_{21} = M_C + M_D \\ state: & \begin{bmatrix} s_{I2} + M_C = s_2 & (state \quad sent) \\ s_1 + M_D = s_F \end{bmatrix} \end{bmatrix}$$

The joining member must process its own pending messages and the state message to build the same final state:

$$g_3 : \begin{bmatrix} messages: & Mnp \oplus M_3 = Mnp \oplus M_{31} \oplus M_{32} \quad (M_{31} \approx M_{21} \quad M_{31} \approx M_{21}) \\ state: & \begin{bmatrix} from \quad g_1: & s_O + (Mnp \oplus M_3 \oplus ms_1) = s_F \\ from \quad g_2: & s_O + (Mnp \oplus M_3 \oplus ms_2) = s_F \end{bmatrix} \end{bmatrix}$$

In general, for an undefined number of members, we can write for a state member:

$$g_i \,/\, i \le m : \begin{vmatrix} Mnp \oplus M_i = M_A + M_B ; s_{Ii} + M_A = s_i ; s_i + M_B = s_F \\ s_i + M_B = s_O + ms_i + M_B = s_O + ms_i + \overline{(Mnp \oplus M_i)} = s_F \end{vmatrix} \quad [\delta 1]$$

In addition, for a joining member, the following equation must be verified, whatever is the coordinator sending the state:

$$g_j \,/\, j > m, \forall i \le m : \begin{vmatrix} M_j = \sum_k M_{jk} = \sum_{k \ne i} M_{jk} \oplus M_{ji} \approx \sum_{k \ne i} M_{ik} \oplus M_{ji} = M_i \oplus M_{ji} \\ s_O + (Mnp \oplus M_j \oplus ms_i) = s_O + (Mnp \oplus M_i \oplus M_{ji} \oplus ms_i) = s_F \end{vmatrix} \quad [\delta 2]$$

Putting together these two equations, the following condition must be achieved:

$$\forall i \le m : \quad s_O + ms_i + \overline{(Mnp \oplus M_i)} = s_O + (Mnp \oplus M_i \oplus M_{ji} \oplus ms_i) \quad [\delta 3]$$

This is the general state transfer condition: it must be possible to build the state from a set of messages and a state message whose included state has been reached after having processed a subset of those messages. It must be taken in consideration that the order on which those messages and the state message is received depends on the application ordering restrictions, and, except when using total ordering, this order cannot be predicted at all.

An example is a group handling a list of clients whose only update is the operation '*add client if not yet existing*', having the state message the whole list of clients. On the other side, if this group would also allow operations like '*remove client*', the previous condition would not always be verified. Be the case where two messages are sent to the group, one adding a client and the second one removing the same client. A member could process the first message and send afterwards the state message, which would include that client. Later, the second message would be processed, finishing in a state where the client is not present. The joining member could process first both messages and receive then the state message, resulting in a state where the referred client is present!

When the state transfer does not inhibit the sending of new messages, the condition to achieve is very similar:

$$\forall i \le m : \quad s_O + ms_i + \overline{(Mnp \oplus M_i \oplus M)} = s_O + (Mnp \oplus M_i \oplus M_{ji} \oplus ms_i \oplus M) \quad [\delta 4]$$

This can be simplified to [δ5], that can be also understood as [δ4] when the blocking period before the view change is so short that no member processes or sends any message:

$$M' \equiv Mnp \oplus M_i \oplus M_{ji} \oplus M \quad \Rightarrow \quad \forall i \le m: \quad s_O + ms_i + \overline{M'} = s_O + (ms_i \oplus M') \quad [\delta 5]$$

It must be highlighted in the previous formula that the subset M' of messages in the left side is composed by those messages in M' not yet processed when the state was sent, but the whole set of messages along with the state message must be processed on the right side. Additionally, the state message should be processed between any two messages in this set, without modifying the final state reached. A possibility is to include in the state message information on the messages already processed.

An interesting case occurs when the state is sent immediately after the view is installed. The other limit case, to send it when every pending message has been processed is not so interesting, because it is hard to implement in an asynchronous system (how to know that there are no messages left?); additionally, no new messages could be sent in the group while the state is being transferred. When the state is sent immediately after the view change notification:

$$\left.\begin{array}{l} M'' \equiv \overline{M'} \equiv Mnp \oplus M_i \oplus M \\ M' \equiv M_{ji} \oplus \overline{M'} \equiv M_{ji} \oplus M'' \end{array}\right\} \quad \forall i \le m: \quad s_O + ms_i + M'' = s_O + (ms_i \oplus M_{ji} \oplus M'') \quad [\delta 6]$$

$$if \quad M_i = \phi \quad \forall i \quad \Rightarrow \quad \forall i \le m: \quad s_O + ms_i + M'' = s_O + (ms_i \oplus M'') \quad [\delta 7]$$

The last simplification [δ7] is valid for groups having only *loopback* messages. It means that there must be no difference on processing first the state message and later a set of messages, or processing the whole set of messages with the state message between any two messages in that set. An example of group accomplishing [δ7] is a counter where every message has a relative meaning ('*increment counter*', '*decrement counter*') and the state message is also understood as a relative increment ('*increment counter by value*'). Even this oversimplified example does not verify [δ5], because the state message could be sent after processing some of the '*increment counter*' messages in the current view, messages that will be processed as well by the joining member, resulting in different final counters.

Previous conditions are difficult to achieve for generic groups. Even the easiest condition [δ7] has no general solution, because no group ordering can make the joining member to receive first the state message and later the rest of messages, what would solve directly the equation!

We propose the following solution to validate the state transfer for any kind of group messages and state messages: the state message is sent immediately after a view change, and the joining member blocks any message until the state message is received and processed. Once it is processed, the blocked messages are unqueued and processed in the input order. If the group has no-*loopback* messages, it is also needed to discard M_{ji}, the messages that where already processed by the coordinator on the previous view. In this case, we have:

$$s_O + (ms_i \oplus M_{ji} \oplus M'') = s_O + ms_i + (M_{ji} \oplus M'') = s_O + ms_i + M''$$

Moreover, the required condition [δ6] is achieved. Because only those messages sent by the coordinator *before* the state message (that is, those sent on the previous view) must be discarded, this solution implies that the messages ordering must be, at least, *FIFO* (only if *no-loopback* messages are used in the group). And because a group could use both *loopback* and *no-loopback* messages, it is an application decision to select the messages (those being no-*loopback*) to discard.

The previous equations (for any kind of group) do not change if there is more than one joining member, having no difference if there is one coordinator for each joining member, or one for all of them and the transfer is done concurrently or sequentially. If sequentially, the solutions based on the state being sent immediately after a view change will require now that the coordinator will not process any message until every transfer is finished.

4.1. View changes

If there is any view change before the coordinator sends its *state message*, the condition to be verified by a generic group is still the one without view changes [δ5]. To demonstrate this, we will suppose that after the state message is sent, no member will send any more messages (to reach therefore the same final state) and no new views are installed in the group. The joining member will have to process a set of messages *M*. It will first process a subset of *M*, then process the state message and finally the rest of messages in *M*:

$$messages: \quad M = M_A + M_B \qquad state: \quad s_O + M_A + ms + M_B = s_F$$

But, if the group verifies [δ5], the order on which the state message is processed must not modify the outcome:

$$s_O + M_A + ms + M_B = s_O + ms + M_A + M_B = s_O + ms + M = s_F$$

And the state message includes a state built after having processed a subset of the messages in M, just like the solution in [δ5]. Therefore, no new restrictions are needed if there are view changes before the state message is sent, even if the coordinator falls down and other state member becomes the new coordinator.

Our proposal to solve the state transfer on generic groups not achieving [δ5] means that, due to the message buffering, the members involved in the transfer can be seen by the rest of members in the group as slower members. More important is that, when a view change is produced before the transfer has finished, this solution violates the virtual-synchronous multicast delivery property. This property states that any pair of processes that are both members of two consecutive group views must *receive* the same set of multicasts during the period between those views. When the view is going to change, both the coordinator and the joining member have messages queued that cannot process, because the transfer has not yet finished. In fact, the property is violated if the messages are not received; however, although not processed, they are received, and the groups must take this possibility in account. The system can be modeled in two ways:

1. The coordinators and joining members are considered as excluded temporarily of the view. They continue blocking the messages, and only when the transfer has finished, the queued messages will be processed. They would proceed probably from different views, but they are still stored in the right order.
2. The joining member is not considered to be in the view while the state has not been transferred. In this case, the coordinator would cancel the transfer and would process the queued messages, restarting the transfer on the new view. The joining member must discard the queued messages if it receives a view installation event before the state message arrives. Because the coordinator could have sent the state message before realizing that the view was blocked (synchronization problem), the joining member must be able to discard this message, which will arrive during the next view. This discarding can be achieved by including the view identity on the state message.

Note that both solutions are excluding, at least, the joining member as being a real member of the group. This is equivalent to a solution where the state is sent to a member not belonging to the group, and only when this transfer has finished, the new member joins the group.

If a new view is installed and the coordinator of an unfinished transfer falls down, other state member must take that role and do the transfer. Nevertheless, our proposal, as happened with the simplifications [δ6,7], assumes that the state message is sent before the coordinator processes any message of the new view. The joining member expects therefore a message with the state of the coordinator after the view installation. If this coordinator falls down, any new coordinator should send that state. To achieve this new condition, there are several possibilities:

1. State members are saving their state after each view change. Moreover, because two members could join on consecutive views, they should store several states.
2. Or they freeze their states by buffering every message until the member gets the state: every member in the group is blocked.
3. Or every state member behaves as coordinators, sending its state once installed a view with new members.
4. A last and preferred solution is that the joining member discarded any messages buffered during the previous view: just as if the member would be joining the group on the last view. This solution is the same as the second possibility we had pointed before for the modeling of the system, and it is therefore our favorite one.

Finally, it must be considered the case where the view is installed before the coordinator sends its state, and it must coordinate as well the transfer to new members on the latest view. Each member expects the state as it is before processing any message of the current view: the coordinator before starting the new transfer will have to process the messages buffered from previous views. From an implementation point of view, two queues are therefore needed, one to queue the messages on the current view and other for previous views.

4.2. Large state transfers

Large states could require splitting the state message in several chunks. The joining member must receive now every chunk to build its state, and different coordinators could send each chunk. Under these circumstances, the new member must satisfy:

$$g_j / j \geq m : s_O + \left(Mnp \oplus M_j \oplus ms_A^1 \oplus ms_B^2 \oplus \ldots \oplus ms_U^p\right) = s_F \quad [\delta 8]$$

That is, the final state is built from the state messages and the list of pending messages; but the state messages could be sent from different state instances and could already include some of the messages that the joining member must process. This is a general condition difficult to achieve by a generic group. An example is a group that stores every group message, and initializes a joining member by sending those messages; but in this case, every state message is referring to the same state instance, that is, $s_A = s_B = \ldots = s_U$.

Even when there is only one coordinator, the previous condition is not simplified, as the chunk state messages could refer to different state instances (the coordinator can evolve due to the incoming messages). If every chunk would refer to the same state, the solution would be easier. As we saw in the normal transfer, waiting to process every message is a difficult solution, and we will better focus on the case when no messages have been processed. In this case, it must be noticed that there is no difference between having only one coordinator or several of them if the messages are *loopback* ones, as every coordinator would have the same state to send. The coordinator starts on its initial state and will evolve to the final one by processing every pending message (Mnp and M_i); the joining member processes its pending messages (Mnp and M_j) and the chunk state messages to achieve the same final state. If there is just one coordinator:

$$\left.\begin{array}{c} \left.\begin{array}{c} \underline{coordinator} : g_i / i \leq m : \\ \\ \end{array}\begin{array}{c} s_{Ii} + \left(Mnp \oplus M_i\right) = s_F \\ s_O + ms_{Ii} = s_{Ii} \\ ms_{Ii} = ms_{Ii}^1 \oplus \ldots \oplus ms_{Ii}^p \end{array}\right\} s_O + \left(ms_{Ii}^1 \oplus \ldots \oplus ms_{Ii}^p\right) + \left(Mnp \oplus M_i\right) = s_F \\ \underline{joining} : g_j / j > m : s_O + \left(Mnp \oplus M_j \oplus ms_{Ii}^1 \oplus \ldots \oplus ms_{Ii}^p\right) = s_F \\ M_j = \sum_k M_{jk} = \sum_{k \neq i} M_{jk} \oplus M_{ji} \approx \sum_{k \neq i} M_{ik} \oplus M_{ji} = M_i \oplus M_{ji} \end{array}\right\} \Rightarrow$$

$$\Rightarrow \quad s_O + \left(ms_{Ii}^1 \oplus \ldots \oplus ms_{Ii}^p\right) + \left(Mnp \oplus M_i\right) = s_O + \left(Mnp \oplus M_i \oplus M_{ji} \oplus ms_{Ii}^1 \oplus \ldots \oplus ms_{Ii}^p\right) \quad [\delta 9]$$

If several coordinators are transferring the state, M_i must be empty for every member:

$$\left.\begin{array}{c} \left.\begin{array}{c} \underline{coordinators} : g_i / i \leq m : \\ \\ \end{array}\begin{array}{c} s_I + Mnp = s_F \\ s_O + ms_I = s_{Ii} \\ ms_I = ms_I^1 \oplus \ldots \oplus ms_I^p \end{array}\right\} s_O + \left(ms_I^1 \oplus \ldots \oplus ms_I^p\right) + Mnp = s_F \\ \underline{joining} : g_j / j > m : s_O + \left(Mnp \oplus ms_I^1 \oplus \ldots \oplus ms_I^p\right) = s_F \end{array}\right\} \Rightarrow$$

$$\Rightarrow \quad s_O + \left(ms_I^1 \oplus \ldots \oplus ms_I^p\right) + Mnp = s_O + \left(Mnp \oplus ms_I^1 \oplus \ldots \oplus ms_I^p\right) \quad [\delta 10]$$

These two equations look as simple complications on the formulas obtained for only one chunk state (δ6 & δ7): the order on which any pair of messages, including the state chunk messages, is processed cannot modify the outcome.

There is a special case when the state and messages are defined in such a way that each message can only modify one of the state chunks. If there are *p* chunks, and *l* is a specific chunk, we can see for each chunk, from [δ8]:

$$\left.\begin{array}{c} \left[s_O + \left(Mnp \oplus M_j \oplus ms_A^1 \oplus ms_B^2 \oplus \ldots \oplus ms_U^p\right)\right]^l = s_F^l \\ \forall j \neq l \rightarrow ms_X^j = \phi \end{array}\right\} \quad \Rightarrow \quad s_O^l + \left(Mnp^l \oplus M_j^l \oplus ms_i^l\right) = s_F^l \quad [\delta 11]$$

[δ11] must be verified on each chunk *l*, when the state is received from any coordinator *i*, that could be different for each chunk. This coordinator has processed a part of the incoming messages before sending its chunk message:

$$g_i / i \le m: \quad \left.\begin{array}{c} messages: Mnp \oplus M_i = M_A + M_B \rightarrow M_B = \overline{(Mnp \oplus M_i)} \\ state: s_{Ii} + (Mnp \oplus M_i) = s_F \rightarrow s_{Ii} + M_A = s_i; s_i + M_B = s_F \end{array}\right\} \quad s_i + (Mnp \oplus M_i) = s_F \quad \Rightarrow$$

$$\left.\begin{array}{c} \Rightarrow chunk: s_i^l + (Mnp^l \oplus M_i^l) = s_F^l \\ s_O^l + ms_i^l = s_i^l \end{array}\right\} \quad s_O^l + ms_i^l + (Mnp^l \oplus M_i^l) = s_F^l \quad [\delta 12]$$

And comparing now [δ11] and [δ12], we obtain an equation like [δ3], but that must be achieved for each chunk, having perhaps different coordinators on each chunk transfer:

$$s_O^l + ms_i^l + \overline{(M_{np}{}^l \oplus M_i^l)} = s_O^l + (M_{np}{}^l \oplus M_i^l \oplus M_{ji}^l \oplus ms_i^l) = s_F^l \quad [\delta 13]$$

If messages are not blocked while the transfer is being done, they must just be included in the previous equation:

$$s_O^l + ms_i^l + \overline{(M_{np}{}^l \oplus M_i^l \oplus M^l)} = s_O^l + (M_{np}{}^l \oplus M_i^l \oplus M_{ji}^l \oplus M^l \oplus ms_i^l) = s_F^l \quad [\delta 14]$$

To obtain a general solution for any kind of groups, we extend the solution proposed for transfers in one state. Messages must be now blocked on the joining member until the last state message is received, and the application must decide which is this last state message. If there are no *loopback* messages, every message coming from the coordinator with lower timestamp (*FIFO* order is therefore required) than the last state message must be discarded. This solution means that only one member is to be used as coordinator for a specific transfer. However, because the coordinator cannot process any message before having completed the transfer, it has sense to use only one coordinator, and not to block several members.

4.3. View changes on large state transfers

When the state of a group can be decomposed into several parts, and messages are designed to update exclusively one of those parts, the state transfer problem can be decomposed in *n* one-chunk state transfers, where each chunk corresponds to each of the state parts. In this case, these groups share the same problems under view changes as we already discussed in the previous section. On generic groups achieving [δ8], that condition does not become more complex (even), although the state members must now consider the case where the coordinator(s) falls down before ending the transfer.

The solutions presented on the previous point based on the sending of the coordinator's state before processing any further message, can use under view changes the same strategies pointed out for one-chunk transfers. Related to our preferred solution, where the joining member discards any previous messages, note that it must also discard the previous state messages! This is an expensive solution: the state is being split to make faster (and perhaps feasible) the transfer, but it must be restarted each time the coordinator falls down. Even supposing that repeated coordinator crashes shouldn't be usual at all, a bi-directional protocol between the joining member and the coordinator would help the transfers, allowing also the transfer of static information through the coordinator.

5. Conclusions

We have shown that only the simplest systems can perform a basic state transfer without doing some extra work, notably the blocking of messages. A general approach to solve the state transfer problem is:

1. The state message is sent immediately after a view change, and the joining member blocks any message until the state message is received and processed.
2. After processing the state message, blocked messages are unqueued and processed in the input order. If the group has no-*loopback* messages, it needs to discard those messages that where already processed by the coordinator on the previous view. In this case, *FIFO* ordering is required.
3. The joining member is not considered to be in the view while the state has not been transferred. When a new view is installed, the coordinator stops the transfer, processing any buffered message. The joining member discards those buffered messages, waiting for a new transfer on the next view. The state message would include the view identity, to be discarded in the case of being received on the next view.

4. If the state message is sent in several chunks, messages must be blocked on the joining member until the last state message is received, and the application must decide which is this last state message. If there are *no loopback* messages, every message coming from the coordinator with lower timestamp than the last state message (*FIFO* order is therefore required) must be discarded.

5. If the state is sent in several pieces and the coordinator falls down, the basic solution is that the joining member will discard the state messages. The performance penalty to pay in this case can be avoided by using a bi-directional protocol on the transfer; the joining member is therefore able to transmit to the new coordinator its state. This solution is not universal, but could avoid the need to re-send the state messages.

A bi-directional protocol can be useful when the state is transferred in several chunks, and therefore the whole state transfer could be modeled to have an initial transfer of information from the joining member to its coordinator. This information does not have just to be its transfer state, but also some static information could be transferred in this way between members.

The second section discussed the state transfer mechanisms for some of the existing group communication systems. When the state was transferred from an active replica, the whole system was generally blocked while the transfer was performed, although some systems just block the involved replicas. Moreover, generally only one step was allowed. We have shown how this block must be performed, depending on the kind of message ordering, and its extension when several steps are convenient due to the state size. We have not covered the state recreation from a log file, like the checkpoint strategy followed by *Cactus* or designed in CORBA. Although this paper does not discuss implementation details like the coordinator election or how to handle a group without state members, these issues are solved and implemented, together with the protocol described in this paper, as a package of C++ classes called *Sensei* [Pena99], built on top of *Ensemble*. This implementation does take into account the previous implementation details, allowing a huge flexibility for the design of reliable distributed applications. It also specifies the higher-level bi-directional protocols that allow a transfer of information between joining members and their coordinators, considering both *pull* and *push* transfers (depending on the information being requested by the joining member or offered by a state member).

References

[Arjuna] http://arjuna.ncl.ac.uk/

[Babaoglu95] O. Babaoglu, R. Davoli, A. Montresor, *Group Membership and View Synchrony in Partitionable Asynchronous Distributed Systems: Specifications*, University of Bologna, Technical Report UBLCS-95-18, November 1995.

[Birman87] K. Birman, T. Joseph. *Exploiting Virtual Synchrony in Distributed System*, Proceedings of the Eleventh Symposium on Operating Systems Principles, pp 123-138. Austin, November 1987.

[Cactus] http://www.cs.arizona.edu/cactus/index.html

[Chandra91] T.D. Chandra, S. Toueg. *Unreliable Failure Detectors for Reliable Distributed Systems*, Proceedings of the Tenth ACM Symposium on Principles of Distributed Computing, pp 325-340, Montreal, August 1991.

[Friedman95]R. Friedman, R. Renesse, *Strong and Weak Virtual Synchrony in Horus*, Cornell University Technical Report TR95-1537, August 1995.

[Electra] http://www.softwired.ch/people/maffeis/electra.html

[Ensemble] http://www.cs.cornell.edu/Info/Projects/Ensemble/index.html

[Hayden98] Mark Hayden *The Ensemble System* Cornell University Technical Report, TR98-1662, January 1998.

[JavaGroups] http://www.cs.cornell.edu/Info/People/bba/javagroups.html

[Lamport78] L. Lamport. *Time, Clocks, and the Ordering of Events in a Distributed System*, Communications of the ACM, Vol 21, Number 7, July 1978.

[Maestro] http://simon.cs.cornell.edu/Info/Projects/Ensemble/Maestro/Maestro.html

[Maffeis97] S. Maffeis and D. C. Schmidt *Constructing Reliable Distributed Communication Systems with CORBA*. IEEE Communications Magazine 14(2), February 1997.

[Malloth96] C. Malloth. *Conception and Implementation of a Toolkit for Building Fault-Tolerant Distributed Applications in Large Scale Networks* PhD Thesis No. 1557, Swiss Federal Institute of Technology of Lausanne (Switzerland) (Ecole Polytechnique Fédérale de Lausanne) September 1996.

[OMG98] *Fault Tolerant CORBA Using Entity Redundance*, OMG Document orbos/98-04-01, http://www.omg.org

[Parrington95] G.D. Parrington, S.K. Shrivastava, S.M. Wheater and M.C. Little. *The Design and Implementation of Arjuna*, USENIX Computing Systems Journal, Vol 8, No 3, 1995

[Pena99] L.M. Pena, J. Pavon. *Sensei: Transferencia de Estado en Grupos de Objetos Distribuidos*, Computacion y Sistemas Vol.2 No. 4, pp.191-201, April 1999.

[Pena99tr] L.M. Pena, *Transferencia de Estado Bajo el Modelo de Sincronia Virtual Debil*, Universidad Complutense de Madrid, Technical Report, November 1999.

[Phoenix] http://lsewww.epfl.ch/projets/phoenix/index.html

[Ricciardi93] A. Ricciardi, A. Schiper, K. Birman. *Understanding Partitions and the "Non Partition" Assumption*, IEEE Proc Fourth Workshop on Future Trends of Distributed Systems, Lisbon, September 1993.

[Schneider90] F. Schneider. *Implementing fault-tolerant services using the state machine approach: A tutorial*, ACM Computing Surveys, 22(4):299-319, Dec 1990

[Schlichting93] R. D. Schlichting, S. Mishra, L. L. Peterson. *A Communication Substrate for Fault-tolerant Distributed Programs*, Distributed System Engineering, vol. 1, pp. 87-103, December 1993