A Flexible API for State Transfer in the JavaGroups Toolkit

Bela Ban
Dept. of Computer Science
Cornell University
bba@cs.cornell.edu

Abstract

When a group member needs to retrieve the group state, it usually solicits it from one or more of the existing members and sets its local state accordingly. There are various criteria for state transfer, such as pull or push, size of the state, blocking or non-blocking etc. We will examine the different criteria for state transfer, and present the API and protocol used in JavaGroups [Ban, Ban98] to transfer state. We focus on how the API and implementation provided can be customized to different application needs.

1 Introduction

This paper discusses state transfer in the context of *process groups* using the *virtual synchrony* model. A process group is an entity which processes join for the purpose of exchanging messages. In contrast to one-to-one communication usually encountered in distributed systems, groups offer one-to-many communication, in which a message sent to the group address[1] is received by all members currently in the group.

An important part of group communication is membership management, that is keeping track of who's currently in the group. Membership may be dynamic, e.g. new processes can join a group at any time, old members may leave, or crash[2], in which case they must be excluded from the group by the membership service.

Virtual synchrony offers a model in which each member has a list of members currently in the group (a *view*). Whenever the membership changes, a new view is installed in all members. Views are uniquely identified by the issuer of the view and a sequence number. There is usually one designated member that broadcasts new views (the coordinator). Every non-faulty group member will receive the same sequence of views. At the core of virtual synchrony is the guarantee that (1) all members receive the same set of messages between consecutive views and (2) a message *sent* in view V1 will be *received* in V1. Virtual synchrony does not make any guarantees about the order of messages within views: to for example deliver all messages in the same order at all receivers, one would have to employ another model (total order) in addition to virtual synchrony. For a more detailed discussion of process groups and virtual synchrony see [Bir96]

Group communication often involves members maintaining a shared state, where every member has the same state, e.g. in a replicated directory service where every member maintains its own local copy of a directory. Updates are sent to all members and applied in the same order, modifying the local states identically. When a new member joins, it needs to obtain its initial copy of the directory from an existing member before it can become operational.[3]

[1] Also called *broadcasting* or *multicasting*.

[2] A crash will eventually be detected by a *failure detector* which leads to exclusion of that process. The effect is the same as if that process left the group voluntarily.

[3] When members are deterministic, that is the same input causes the same behavior (without any external stimuli) in all members, this approach is called the *state machine approach* [Sch86].

It is also possible that, in some groups, only a few designated members maintain (share) the state and synchronize state changes among each other, and the other members do not maintain any state.

Depending on whether the state is shared among all members, or only among a small subset, a new member joining the group may want to retrieve the state from an existing member, from several members or from all the group members. The first case would typically be used when all members share the state: in that case, knowing that all member states are guaranteed to be identical, the group state may be retrieved from just one member (e.g. the oldest). Contacting more than one of the existing members may be used for security reasons (e.g. no tampering with the state by a single member) or to ensure, by comparison of multiple states, that the state retrieved is correct. Retrieving the states of all members may be used when the member states are disjunct: this may be useful in a load-balancing application when a new member wants to inform itself about the progress of the other workers in the group by looking at their states.

This paper discusses the various aspects of state transfer and its consequences on the design of a state transfer API. As each application has different requirements on state transfer, we believe that there cannot be a single implementation that fits all needs. However, we will argue that a) different *implementations* of a state transfer API can be hidden in *protocol layers* and b) different state transfer APIs can be offered by different components (or objects) layered on top of a simple group communication API.

We will start by looking at a (non-exhaustive) list of aspects of state transfer, and derive from it the requirements for a flexible state transfer API. The design and implementation of the API are then discussed in the context of a Java based toolkit for reliable group communication, *JavaGroups* [Ban98].

2 Criteria for State Transfer

2.1 Roles

Participants of a state transfer can be divided into three categories: (a) state requesters, (b) state providers and (c) non-participants. State requesters are usually new members (clients) that need to acquire the current group state to become operational members. Non-operational members (clients) cannot provide state yet. Operational members can choose to provide state or not. When they do, they are potential state providers (servers), and when they don't, they are non-participants. The division between servers and non-participants is important because it permits operational members to ignore state transfer requests, therefore being able to serve other tasks. The choice whether to serve state requests or not should be reflected in the API and should be modifiable at runtime, therefore allowing a state-serving member to become a non-participant and vice versa. A possible setup may for example have a small subset of group members that serve state requests (all state transfer requests have to be directed to one or all members of that subset) and – when one of them leaves or crashes – a non-participant would become state serving.

2.2 Sharing

The degree of state sharing depends on the type of application involved. In the *state machine* approach [Sch86], all members receive the same sequence of messages which trigger the same state transitions in all members. Members have to be deterministic, i.e. their state transitions depend solely on messages received, and there may not be any sources of non-determinism, such as threads, semaphores, shared memory or timers. As each message advances the member states in the same manner, all members have the same state.

When a new member joins a group consisting of replicated servers, it will need to contact only a single existing member to obtain the shared group state (typically the oldest member). As soon as it has integrated the state, it will become a server and is able to pass its state on to others.

At the other extreme of state sharing are groups without (or with minimal) shared state, e.g. load balancing groups, where each member works on a subset of the problem and maintains state only for its specific task. A coordinator typically divides the problem space and assigns each member a different slice. Then it waits for completion of all tasks, re-assigning tasks to new (or finished) members if existing members crash, and combines the partial results. In this case, a new member joining the group does not need to acquire any shared group state, but will typically be assigned its initial state by the coordinator.[4]

Between the two extremes of sharing all the state and sharing no state at all, there may be groups that only share a minimal state, e.g. for coordination purposes, or groups where a subset of the members share some state. In the latter case, a mechanism must be provided by the state transfer API to obtain the state from a specific *member* or a *set of members* (see 2.3).

If a process is member of multiple groups, the degree of sharing state may vary with each group (see 2.11).

2.3 One or Many State Providers

A state transfer API should provide functions to retrieve state from a single member, or from a set of members where the set may include all group members.

With such a mechanism, a new member may retrieve the initial state from a single member, e.g. the oldest member, or from a number of group members, e.g. to make sure that the state is correct by comparing the results and incorporating the state of the majority. For example, if 3 members return states S2, S2 and S1 respectively (where the numbers signify state versions), then the quorum state S2 would be assumed to be the most recent state.[5] The selection of members from which to solicit state may be highly application-specific, e.g. obtain state from the fastest servers in the group, or from the 3 servers that provide state (whereas the rest of the members don't).

2.4 Time

The time when state can be acquired may vary. A group communication toolkit may return the current group state to the new member when it joins the group. A new member indicates (by setting a flag in the JOIN request) whether it wishes to receive the current group state. If set to true, the state is returned either as result of the JOIN request, as first message the new member receives, or as part of the first view installed. Returning state with a view change avoids having to account for messages sent *during* the state transfer: members typically do not send messages during view changes anyway[6], therefore this is an ideal time for state transfer (see 2.7 for a discussion of blocking vs. non-blocking state transfer).

However, in some settings, it is the *application* rather than the toolkit that may want to specify the time of state transfer: one could think of applications that obtain the shared group state more than once (repeatability), or control members that regularly fetch all member states, compare them and – if not all states are equal – initiate a state correction protocol that installs the correct state[7] in all members having incorrect state.

Separating state transfer from the JOIN API and making it an independent abstraction results in greater flexibility: a new group member may choose to acquire state when joining, and/or it may do so at any time afterwards.

---

[4] It is assumed here that when a member crashes, a new member will have to start work on a task from scratch. However, if it takes a long time to complete a task, other schemes such as making the state persistent, or having a backup member which is initialized to the same state, are possible. In this case, state transfer would need to be done from the primary worker to the backup worker (cf. 2.10).

[5] In certain applications where the member states always have to be the same, such a scenario is regarded as an error and may trigger a synchronization protocol to make all members agree on a new shared state.

[6] E.g. after a FLUSH protocol [Bir96, ch. 13.12.4] has been run.

[7] Correctness e.g. based on quorum state.

Duration of state transfer is also an important aspect: if a state transfer implementation involves blocking other group members, then a long duration (e.g. due to a large state) is detrimental to group availability.

2.5 Active or Passive

There are essentially two ways of obtaining state: the application may actively *pull* the state from one or more group members, or the group communication toolkit may *push* the state to the application when it sees fit. An example of the latter is when a network partition heals and a group G1 is merged with another group G2: in such a case, the toolkit has to select (e.g. based on state version numbers, or primary partition flags) the group whose state is the one to which the combined group's state should be set. If it was G2, then all members of G1 would be updated with G2's state by the group communication toolkit. Another example of pulling vs. pushing state is when a new member joins a group: adopting the pull-mechanism, a new member would retrieve the state from one or more members, whereas a push-mechanism would involve a number of existing members (possibly all) pushing their state to the new member as described in [Bir96, ch. 15.3.2]. Note that the latter approach is redundant when all members share the same state and may not scale well with group size.[8]
A pull-approach tends to give more flexibility to the application, e.g. the time of state transfer can be determined by the application, whereas a push-approach may sometimes be needed to update members incorporating an incorrect state (e.g. after merging network partitions). However, the two approaches can be combined: a member's state is updated whenever triggered by the toolkit, and it in turn may actively retrieve the state whenever necessary.

2.6 Size of State

The size of the state to be transferred influences both the API and the implementation chosen for state transfer.
A protocol implementation that requires blocking (see 2.7) should avoid blocking for a long time, therefore it is more suitable for transfer of small state. An implementation that only blocks the receiver of state with respect to all other members during state transfer is not susceptible to blocking for a long time, as the receiver is not yet a group member and providing any service, therefore it is more suitable for transferring larger state.
The choice of API heavily depends on how a large state is transferred, for example large state may be transferred in multiple messages, or it may be prefetched before joining a group and then only the update state would be transferred, or members may rely entirely on external state, as discussed below.

2.6.1 Transfer in Multiple Pieces

When a state is large, an application may decide to fetch it in multiple pieces, or a toolkit (in the push-approach) may decide to push the state to the joiner in several steps. When a state is transferred in multiple pieces, the end of the transfer is typically signalled, e.g. by a `done()` message as described in [Maf95].
The main advantage of such a scheme is that – if state transfer is performed over the same communication line as regular group communication traffic – the additional load can be kept rather small, depending on the size and transmit interval of the individual pieces.

2.6.2 Prefetching of State

When a state is large, a process wishing to join a group may prefetch the bulk of the group state either from existing member(s) or an external source *before joining the group and becoming*

[8] Joining a large group may lead to message overflow in the joiner due to the large number of state messages sent to it by other members.

operational. At some time after the process actually joined the group, an *update protocol* would retrieve the remaining state, that is the state modifications that were applied after retrieval of the initial state. Thus, by performing the larger work of state transfer offline before joining a group, acquiring state when the member is online would require only a small amount of time, making the new member operational quickly. See section 3.4.4 for an example of an update protocol.

2.6.3 External Shared State

If shared state is too big to be transferred efficiently (clogging up network bandwidth), then a possibility is to store it externally, for example in a shared database. Thus, members would not maintain their own state (or very minimal state), and all state-related access would be directed towards the database. Since updates to a database are serialized, all members would always see the same state. However, having a common database means that every member may potentially want to update the database (reflecting its state) after receiving a message, and – since all members receive the same message (in the state machine approach) – they would all try to apply the same modification to the database, which is redundant. Therefore, sharing state via an external source typically involves some kind of primary–backup scheme (cf. 2.10) where the primary server updates the database, and backup servers are only permitted to read from the database (passive replication). When the primary server crashes, a deterministic mechanism will make one of the backup servers the new primary, without having to transfer state to it in order to start work.
If shared group state is kept in a database, which represents a single point of failure, such schemes typically involve a backup database.

2.7 Blocking vs. Non-Blocking

State transfer may involve blocking the involved members for a period of time, e.g. until the state receiver has successfully integrated the group state. If messages were delivered to the group while a state transfer was in progress, the received state would already be stale and not reflect the up-to-date group state. Therefore, schemes involving blocking all members often allow state to be transferred only with view changes, i.e. at a time when the group is blocked anyway (see [Vay98, Bir96, Maf95]). Vaysburd for example describes a protocol where a *transfer view* is installed for a state transfer, followed by a regular view when the transfer is completed.
Blocking all members during state transfers decreases availability and may lead to slow response times for larger group sizes. This is aggravated when a large state has to be transferred: the time needed for the transfer is the time that all members are blocked. Therefore, blocking of members is more suitable where only small state is shared, or in situations where no state is shared at all, for example load balancing, or state that is stored externally.
An alternative to blocking all group members is blocking the *joiner*: all messages sent to the group are delivered at all members during a state transfer, except at the joining process which queues them until it has integrated the state (by obtaining the latter from a member). Then it applies all queued messages to the state and from now on delivers all messages directly. The advantage is that only a single member is blocked, and since it is not yet operational anyway (because it is not yet a real group member), it does not affect the group service as a whole anyway. All other members are allowed to make progress, thus increasing the availability of the group. This mechanism is described in more detail in section 3.3.

2.8 Concurrency

Multiple state transfers may be running concurrently, initiated by different members at almost the same time. To discern between different states, they have to be tagged with *state version numbers*. As proposed in [Vay98], a version might consist of a major and a minor number. The major number would be the view ID and the minor number the number of updates to that specific state. The state version would have to be specified in each request in the state transfer protocol implementation, but would not necessarily need to be reflected in the API.

2.9 Transfer Medium

Transferring large state over the regular group channel may adversely affect throughput of regular group messages. Therefore some applications may want to open a separate 'out-of-band' channel to transfer large chunks of state without impairing regular messages. Such a channel would be used exclusively for state transfer. It might be created at startup, or whenever state transfer is needed.

2.10 Primary–Backup Replication and Snapshots

In primary-backup replication clients send all requests to a primary server which in turn updates a number of backup servers (all in the same group). When the primary crashes or is shut down, a backup becomes the new primary. There are various methods to keep the backup servers updated: a simple scheme for example is one where the primary multicasts all updates to all members of the backup group. A different scheme involves logging all updates at the primary to stable storage and periodically multicasting snapshots to all backup servers. In case the primary crashes, the log written by the primary after the last snapshot can be used to reconstruct the current state. Yet another solution may have a designated backup server periodically solicit the state from the primary and – when received – distribute it to all backup servers. The latter is an example of a case where state is not transferred to a *joining* process, but to a member that may have been operational for a long time.
In a snapshot approach, the transfer of state from the primary to the backup servers should not take a long time, as the primary will not respond to requests during that period. Therefore, a non-blocking state transfer protocol is preferred in situations involving long transfer times.

2.11 MultiGroups

Members of multiple groups must have a means to differentiate between states from different groups. In the provider role, a member solicited for its state must be able to return the state for the correct group. By the same token, a receiver of state must be able to tell from which group the state was sent and initialize that state accordingly.
A multigroup application may choose to use pull-style for some groups and push for others. Also, sharing of state may vary between the different groups of which it is a member. Hence it is mainly the responsibility of the application to coordinate multiple groups.
In addition to the issue of state transfer, multigroups face a number of other problems such as preserving causal and total order for multigroup multicasts, atomic broadcasts to multigroups and multigroup membership management (e.g. how can a member join a number of groups at once). As these issues are very complex, in-depth treatment cannot be given here. Instead, we refer to [Bir96, ch. 14.2] for details.

3 State Transfer in JavaGroups

This section describes the API and implementation of the default state transfer mechanism provided by JavaGroups. As was discussed previously, requirements on state transfer differ, both in terms of API and implementation. Therefore the task of providing a common API for all different needs is daunting. We adopt the view that there cannot be a single API, covering all different aspects. Hence, our approach is to provide a *minimal API* (integrated into a JavaGroups channel) and *protocol implementation* for state transfer. In 3.4, we will discuss how to replace API, implementation, or both, to adapt to different needs in cases where the default API/implementation is not sufficient. Thus, a solution is provided that is flexible enough to be tailored to different needs and yet provides a certain base functionality so that applications don't need to re-implement state transfer.

3.1 Architecture of JavaGroups

The architecture of JavaGroups is shown in fig. 1. At the top is a channel, which is a simple socket-like group communication endpoint. It has a local address which is attached to every message sent over it so that receivers may send replies, and a group address (usually a name) indicating the group associated with it. Applications send messages via channels to all or single members (multicast/unicast), receive messages, receive notifications when members join or leave the group, get the membership and so on. A channel is deliberately rather primitive so that it fits the needs of various applications. More powerful abstractions can be placed between it and the application to provide more sophisticated functionality.

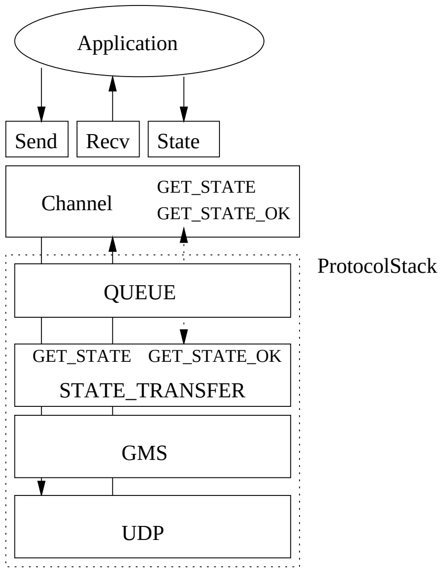

Figure 1: Architecture of JavaGroups

The channel is connected to a protocol stack, the composition of which is defined by the application when creating a channel. The stack essentially consists of a linked list of layers which are connected bidirectionally by two FIFO queues. Layers receive *events* (class `Event`) from their neighboring layers, process them and possibly pass them on to the layer above/below. A special event is a *message* (class `Message`). When the application sends a message, the protocol stack wraps it in an event and passes it down the stack. Each layer may modify, reorder, pass on or discard the event. The bottom layer sends messages to their destination(s). The difference between messages and events is that events never leave the stack, whereas messages (wrapped in events) are unwrapped by the bottom layer and put on the network. When a message is received, it is wrapped into an event and passed up the stack.
Each layer may or may not *respond* to an event, i.e. handle it. Each event has a type and an argument. Depending on the type, a layer determines what to do with the event. The default is to forward it to the next layer.
Layers that respond to certain events may advertize this (e.g. `STATE_TRANSFER` responds to `GET_STATE`). Layers that require that certain layers above/below them respond to certain events, may advertize this too. When a stack is created the protocol stack object checks whether all requirements are matched. If this is not the case, the stack will not be created. For example, layer `STATE_TRANSFER` requires events `START_QUEUEING` and `STOP_QUEUEING` to be handled by

some layer above it. It therefore advertizes this fact (not shown in the figure). Since layer QUEUE advertizes that it handles these two events, stack creation is successful. It would fail if QUEUE was missing.[9]

In fig. 1, the state transfer API generates event GET_STATE and expects response event GET_STATE_OK. It requires a layer below to respond to the former event and reply with the latter to successfully perform state transfer. It does not need to know which layer actually handles the event, and how state transfer is implemented, it only needs to know the format of the state transfer event and that some layer will take care of the rest. This loose coupling will allow us to replace API and protocol implementation independently as discussed in section 3.4.

3.2 API

The API for state transfer is part of the `Channel` interface [Ban98]. A channel can choose whether to serve state transfer requests by setting option `GET_STATE_EVENTS` to true/false (default is false). When state transfer is disabled, the group state can be still received as long as there are other members serving state transfer requests.

The state can be requested by calling `boolean GetState(Object provider, long timeout)` or `boolean GetStates(Vector providers, long timeout)`. The former requests the state from a single member (the oldest member if `dest` is null), whereas the latter retrieves multiple states from a number of members defined in `dests`. If `dests` is null, the states of all members (except the initiator) are returned.

`GetState(s)` returns true or false, depending on whether a valid state could be retrieved. For example, if a member is a singleton, then calling this method would always return false.[10]

After `GetState(s)` has been called, one of the next `Receive` calls will return a `SetStateEvent` object containing the state(s) that the initiator of the state transfer requested.

A state provider (see 2.3) will receive a `GetStateEvent` when calling `Receive`. It should make a copy of its state and return the copy by calling `Channel.ReturnState`. The copy will be stored in the `STATE_TRANSFER` layer until it is actually requested by a client. Note that the state has to be serializable.

The following code fragment shows how a group member participates in state transfers:

```
channel=new JChannel("UDP:PING:FD:GMS:STATE_TRANSFER:QUEUE");
channel.SetOpt(Channel.GET_STATE_EVENTS, new Boolean(true));
channel.Connect("TestChannel");
boolean rc=channel.GetState(null, 5000);

while(true) {
    Object ret=channel.Receive(0);
    if(ret instanceof Message)
        ;
    else if(ret instanceof GetStateEvent) {
        // copy=CopyState(state)
        channel.ReturnState(copy);
    }
    else if(ret instanceof SetStateEvent) {
        SetStateEvent e=(SetStateEvent)ret;
        // set state from e.GetArg();
    }
}
```

[9] Actually, the name of the layer is not important, but *which events it handles*. This allows for example other layers to replace QUEUE if the two events are handled.

[10] A member will *never* retrieve the state from itself !

A channel has to be created whose stack includes the STATE_TRANSFER protocol. Option GET_STATE_EVENTS should be enabled, as the channel might probably want to return its current state if asked. A group called "TestChannelGroup" is joined calling the Connect method. GetState() subsequently asks the channel to return the current state.[11] If there is a current state (there may not be any other members in the group !), then true is returned. In this case, one of the subsequent Receive method invocations on the channel will return a SetStateEvent object which contains the current state. In this case, the caller sets its state to the one received from the channel.

If state transfer events are enabled, then Receive might return a GetStateEvent object, requesting the state of the member to be returned. In this case, *a copy of the current state should be made* and returned using Channel.ReturnState. It is important to a) synchronize access to the state when returning it since other access may modify it while it is being returned and b) make a copy of the state since other accesses after returning the state may still be able to modify it ! This is possible because the state is not immediately returned, but travels down the stack (in the same address space), and a reference to it could still alter it.

3.3 Protocol Implementation

3.3.1 Algorithm

The protocol for state transfer is shown in fig. 2. The process wishing to retrieve the group state broadcasts a MAKE_COPY message to all members (including itself).[12] For the state servers to be able to differentiate between different concurrently running state transfers, the broadcast includes a *state version number*, e.g. the address of the state transfer initiator and a timestamp (P1 in the example).

When the initiating member receives this message, it starts queueing subsequent messages.[13] All other members make a copy of the current state upon reception of the message and tag it with the assigned version number (P1).

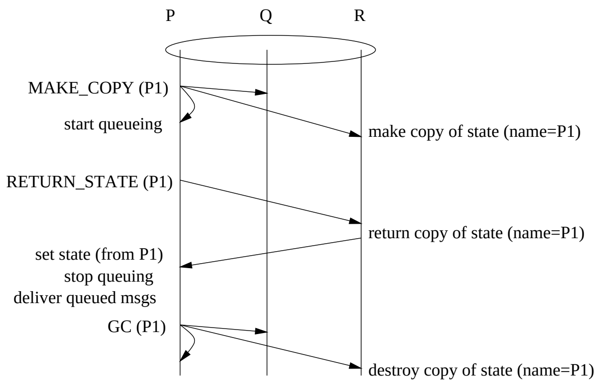

Figure 2: State Transfer Protocol

In the second phase, the initiator selects one member (possibly the oldest, often the coordinator) and sends a RETURN_STATE message to it, including the state version number that was previously assigned (P1). The receiver looks up the copy of the state that was created upon reception of MAKE_COPY associated with P1 and returns it. The initiator sets its state from P1 and then delivers all queued messages before accepting regular messages again. (Message will not be queued after replaying the queued messages).

[11] As the method's destination member is null, the member from which to get the state is selected by the toolkit.

[12] This implies that the initiator already has to be a member.

[13] A member cannot initiate several concurrent state transfers: only a single transfer is permitted to take place at a time.

The last round of the protocol is again initiated by the state retriever and garbage collects the state copies not needed any more by broadcasting a GC message indicating the state version number to be deleted. This avoids having old copies around, taking up memory unnecessarily. The garbage collection broadcast is shown as an independent message, but in a typical implementation, it will probably be piggybacked on another message to reduce broadcast traffic.

This scheme requires that the MAKE_COPY message is delivered at all members at the same position in the global message delivery sequence: the client starting to queue messages and the state servers making a copy of their state are two events that have to take place at the same logical time. This requires the presence of a total order layer in the protocol stack. Fig. 3 shows a problem that could arise if total order was missing: the MAKE_COPY message is broadcast by P at roughly the same time as another message m1 by R.

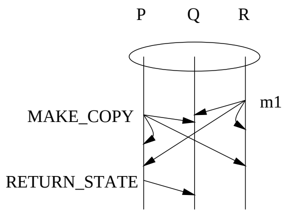

Figure 3: Ordering problem

Whereas both Q and R deliver m1 and then MAKE_COPY, the reception of m1 at P might be slightly delayed, therefore P might deliver MAKE_COPY first and then m1. When Q and R receive MAKE_COPY they create a copy of their current state, including m1. At P, however, m1 is queued as it is seen after MAKE_COPY. When P later retrieves Q's copy of the state and integrates it, the copy includes m1, but subsequent replaying of P's queued messages delivers m1 again. Unless m1 is idempotent, this clearly leads to problems. The situation is even worse if P sees m1 before MAKE_COPY and Q after MAKE_COPY: Q's copy of the state will not include m1, whereas P will see m1, apply it to its state and then retrieve the state from Q. When integrating Q's state, P's own state will be overwritten, thereby losing m1.

A total order layer overcomes these problems by either delivering MAKE_COPY before m1 or after m1 at all receivers. The problem of spurious or lost messages will not occur in this case.

This algorithm is non-blocking: group members (except the initiator) are not blocked during state transfer. The only time an operational member is solicited is when it is asked to return a copy of its state (which will subsequently be stored in the STATE_TRANSFER later (see below). Members who do not support state transfer (see 2.1) will not be solicited at all, not disturbing their processing, and a null state will be returned instead.

3.3.2 Protocol Interaction

The events exchanged between peer STATE_TRANSFER layers and their channel instances are shown in fig. 4.[14]

The protocol on the left side represents the initiator of a state transfer, the one on the right the participant (e.g. an existing group member). The protocol is started when a GET_STATE event is received. It causes 2 messages to be sent to a) all members and b) one selected member.

The first message is MAKE_COPY (1) and it is sent to all members. Upon reception, the *initiator* reacts differently from the participants: it starts queuing all subsequent messages. *Participants* try to retrieve the state from the application. This is done by sending up a GET_APPLSTATE event and waiting for a GET_APPLSTATE_OK event to be returned. If the channel does not support state transfer events (corresponding option is not set), then the GET_APPLSTATE_OK response will

[14]For simplicity, concurrency issues involving state version numbers are omitted.

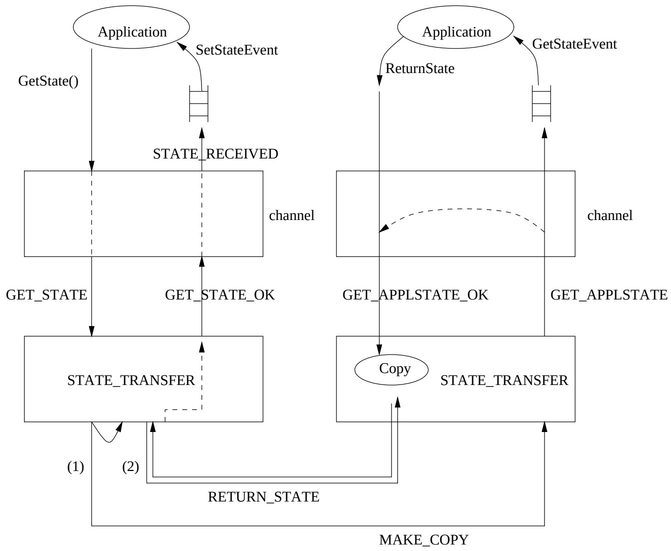

Figure 4: Events between `STATE_TRANSFER layers and channel`

contain a null state. Otherwise it will be a copy of the application's current state.[15] The copy is then stored in the `STATE_TRANSFER` layer for subsequent retrieval by the initiator.
The second message is a `RETURN_STATE` message (2) sent only to one selected member.[16] This is currently the oldest member (usually the coordinator). Note that the message is *never* sent to the initiator itself. The receiver just returns the previously saved copy of the state.
When the initiator receives the copy from the selected member, it does two things: first it sets its own state from the data received. Then, it blocks all message reception, replays the message queue (to which all messages received after `MAKE_COPY` were queued), disables queueing and unblocks message reception. From now on, messages received will not be queued, but passed up to the channel and on to the application where they will be applied to the application's current state. Queueing ensures that any message sent after `MAKE_COPY` and before receiving the state will not be lost, but properly applied to construct a valid state.
When the state has been received, a `GET_STATE_OK` event is passed up the stack. When processed by channel, a `STATE_RECEIVED` event is created and inserted into the channel's event queue. After this, a caller blocked on the `GetState` method will return (true or false). One of the next calls to `Receive` will return a `SetStateEvent` object (generated from `STATE_RECEIVED`) containing the state, which can then be set by the application.
The implementation of `STATE_TRANSFER` relies on total ordering of messages multicast to all group members: it is based on the assumption that all members receive the `MAKE_COPY` message at the same logical time. If this was not the case, then the different participants might save different versions of the state. For example if participant Q received messages m1, `MAKE_COPY` and m2, and participant R received m1, m2 and `MAKE_COPY` then, the state saved by Q would include m1, but R's state would include m1 and m2.

3.4 Customization

The state transfer API and implementation may be sufficient for a number of applications so that they can use it unchanged. However, as there are differing application requirements, some

[15]Note that the data type of the state has to be serializable as it will be marshaled/unmarshaled to be sent over the network. If this is not the case, an exception will be thrown.
[16]If `Channel.GetStates()` was called, then this message will be sent to *all* members instead (excluding the initiator).

applications may want to (1) use a different state transfer API, (2) replace the protocol implementation but keep the API, or (3) replace API and implementation altogether.
The flexible architecture of the JavaGroups protocol stack allows us to do this, as explained in the next sections.

3.4.1 Replacing the Protocol Implementation

If an application wants to use the default state transfer API, but provide its own *protocol implementation*, the `STATE_TRANSFER` (and possibly `QUEUE`, which is used by the former) layers have to be replaced (see fig. 1).
The only event that needs to be handled by the replacement layer for `STATE_TRANSFER` is `GET_STATE`. This event is sent down by the default API whenever `Channel.GetState(s)` is called. It contains information about which member(s) the state is to be retrieved from. When done, the state transfer layer sends the state up the stack in a `GET_STATE_OK` event. The channel will receive the state and return it to the application in one of the next `Channel.Receive` calls.

3.4.2 Replacing the API

The approach chosen in JavaGroups is to provide a minimal API for state transfer (greatest common denominator) and provide hooks to replace/extend it if it cannot be used by the application. The way to do this is to make use of an additional *building block* between the channel and the application, as shown in fig. 5.

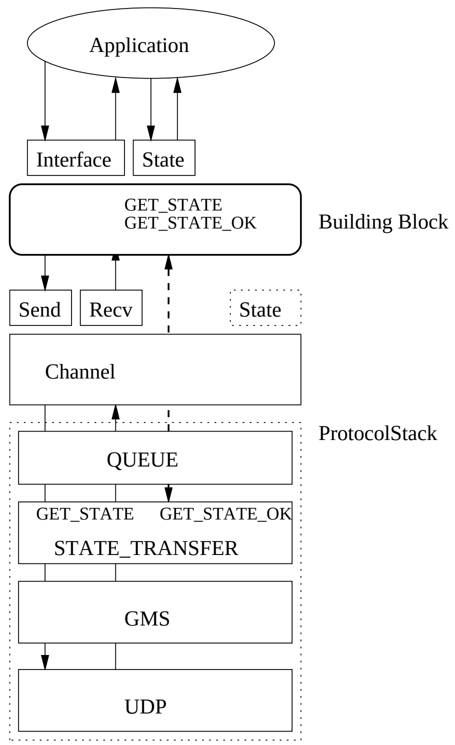

Figure 5: Replacing the default state transfer API

This building block uses the channel for communication, but represents a higher-level abstraction (or just a different API) towards the application. The building block may or may not add functionality, or it may just present the API (interface) differently.
An example of the former is an API that transfers state in multiple, small fragments in order to avoid channel congestion when transferring a large state. The API would send multiple `GET_STATE`

events down the stack to be handled by a state transfer layer, each containing the number of the fragment (plus possibly a state version number for concurrent state transfers) to be transferred. (Note that the current `STATE_TRANSFER` layer would have to be modified to handle fragmented state).

An example of the latter is the `PullPushAdapter` (cf. [Ban]) which converts between the pull and push-approach of fetching messages (see 2.5). The default state transfer API requires the application to retrieve state by calling `Channel.Receive`, state will be returned in a `SetStateEvent` object. The `PullPushAdapter` building block – added between channel and application – converts from a pull to a push-approach: it retrieves messages from the channel and, if the message is a state transfer request or reply, calls the corresponding *callback* (`GetState`, `SetState`) in the application (which has to implement the `MessageListener` interface).

3.4.3 Replacing the API and Protocol Implementation

The default state transfer layer can be abandoned altogether, replacing both the API and the implementation and using one's own mechanisms. This involves (1) providing a building block (on top of the channel) offering a state transfer API which the application uses, (2) replacing the `STATE_TRANSFER` and possibly `QUEUE` layers, (3) generating different events from the building block and (4) handling these events by the corresponding protocol layers that replace the existing ones.

An example where the API would be replaced, the state transfer protocol implementation *discarded* and an existing protocol layer (group membership) modified is given below:

The default implementation of state transfer in JavaGroups does not return the group state to a joiner as result of the JOIN request, as for example described in [Maf95, Vay98, Bir96]. However, one could provide a building block on top of a channel offering a `byte[] Join(Object group_name, boolean get_state` method which optionally returns the state when joining a group. The group membership protocol would have to be modified slightly to make use of the FLUSH phase to obtain and return the group state.[17]

Sliding a building block in between the channel and the application *and* changing the protocol requires that the building block and the protocol 'understand' each other: events sent down by the building block have to be handled by the protocol, so the protocol has to know the types and arguments of these events. By the same token, the building block has to know the types and arguments of response events sent by the protocol. This means that, when the application requests state by calling the building block, the latter will send one or more events down the stack where they have to be caught and handled by the corresponding new state transfer layer. That layer would then send response event(s)s up the stack, through the channel, which would pass them on to the building block, where they would be received.

Thus, the JavaGroups protocol architecture permits to vary API and protocol independently as long as they agree on the same set of events. This allows us to easily extend/adapt the state transfer mechanism to the various needs of applications by either using the default state transfer provided by JavaGroups, or if this is not sufficient, by selecting the right building block that suits the application, or finally, by creating one's own building block and protocol implementation.

3.4.4 Example

This section illustrates how the default mechanism for state transfer in JavaGroups can be adapted. Consider a distributed replicated telephone switch which stores all the state associated with a switch in a database. The frequency of read and write access to the database is high, as new connections are created and torn down at a high rate. To make the switch fault tolerant, there is a backup database, which contains the same state as the primary database. The primary

[17] After running a FLUSH protocol, no more messages are sent until a new view is installed. This guarantees that the state will not be modified as result of message reception.

database would update the backup whenever it receives a modification (passive replication).[18] When the primary database crashes, the backup would be made primary and a new backup created and initialized from the new primary.
Since the primary switch database is large and heavily accessed, blocking it for state transfer is not an option, as it might take a long time to transfer state, during which neither database would be available. Using JavaGroups' default state transfer is not an option either, because the transfer might take a long time, and – as messages sent during transfer are queued at the receiver of the state – there might be a message overflow at the receiver's side.
Therefore, the backup database server *prefetches* the state of the primary database and uses it to initialize its database *before actually joining the group*. The primary would log all requests received during the prefetch phase to both the database and an update log. When the backup server is initialized, it would request the update log from the primary using the regular state transfer, queueing all requests until the update is integrated. At this point, the primary server would delete the update log and the backup server would be operational, replaying the queued request and accepting requests broadcast to the group by the primary.
The protocol is shown in fig. 6.

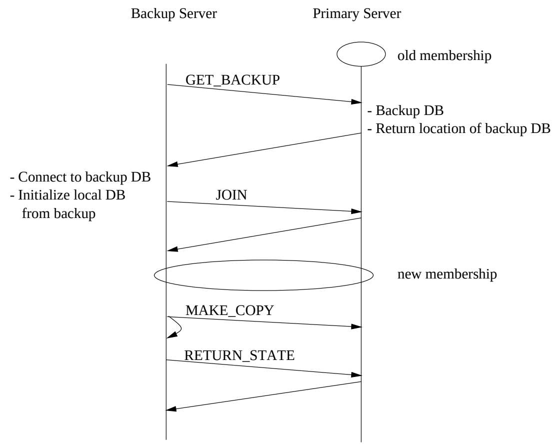

Figure 6: Prefetching the state

When a new backup server is created, it first contacts the primary via a communication link different from the channel[19] (e.g. a separate socket on which the primary is listening) to tell it to create a backup of its database. Today, most modern DBMSs allow to create backups during operation. This is necessary in our case, since we don't want to block requests from being served while creating a backup copy of the primary database.
When the primary server receives the `GET_BACKUP` request, it (a) starts the backup process and (b) at the same time starts logging all requests to an *update log* (e.g. a small database). All the requests received between the backup request and the retrieval of the update log will be written to both the database and the update log. When the primary is done, it returns some information about the location of the newly created backup database to the initiator.[20] The initiator (the backup server) uses this information to locate the backup database and initialize its local database from it.
When the local database is initialized, we need to retrieve the requests sent to the primary in the meantime and apply them to the database. To do this, the default state transfer protocol

[18] Actually, an update would be sent to the group consisting of the primary and backup. This ensures that all updates are performed in the same order.
[19] Not via the group channel, as it is not yet a member. Also, this way the channel is not clogged up with state transfer data.
[20] Note that no requests will ever be applied to the backup database.

can be used. The backup server first joins the group (this is a prerequisite for participating in state transfer) and broadcasts a `MAKE_COPY` to the primary server. Upon reception, the latter stops logging to the update log and returns the update log. This may be the contents of the log, or – as in the backup database itself – the location. The update log will be deleted later. The former starts queueing requests until the update log has been retrieved and applied. When the contents of the update log have been incorporated into the backup server's database, both the backup database and the update log at the primary will be deleted (not shown).
The advantage of the above solution is that the larger part of the state is retrieved offline (outside the normal state transfer protocol), and the transfer protocol is only used to transfer the update state. This avoids a long state transfer and does not overflow the receiver with messages sent during state transfer.
To hide all the protocol interactions, a building block would typically be used (let's call it `Prefetcher`). As shown in fig. 7, it would be layered between the application and the channel.

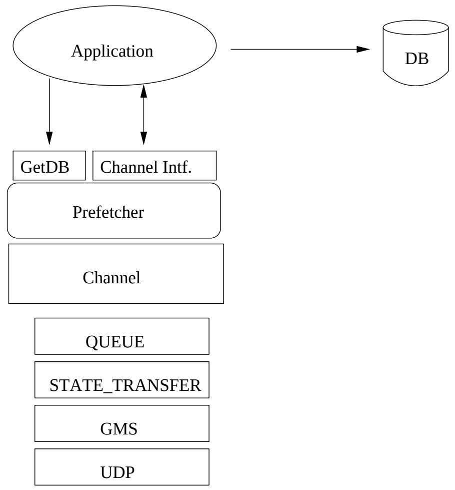

Figure 7: Prefetch Building Block

The application would create a channel with a prefetcher on it, and then exclusively use the prefetcher. The latter manages the channel and offers the same interface as the channel (forwarding requests to it) and an additional method to retrieve a reference to the local database (`GetDB`). This method performs all the protocol interactions discussed above: it uses the channel's state transfer API to retrieve the update log, and provides its own functionality to prefetch the database from the primary. The latter functionality could be implemented in the `Prefetcher` API directly, or it might be added in the form of an additional protocol layer which which the prefetcher would communicate via a new set of events.
The `GetDB` method would contact the primary via an external communication mechanism, initialize its local database, join the group and then use the default state transfer to retrieve the update log. Then it would return a handle (or reference) of the newly created database to its caller (e.g. the backup server) which becomes an operational member and subsequently applies all requests to its local database.
The value of the prefetcher building block is that it hides a number of protocol interactions behind a higher-level and easy-to-use interface tailored specifically to the application at hand.

3.5 Conclusion

We have shown that there are various differing requirements for state transfer in group communication applications. Since state and the transfer of it is often very application-specific, it is useless to provide an API that satisfies all needs. Instead, we provide a minimal default API and a framework that allows application programmers to selectively replace either API or implementation (or both) to suit the needs of their application. Programmers have a choice of the default API, prefabricated building blocks or application-specific functionality to perform state transfer.

3.6 Acknowledgements

The author would like to thank Ken Birman for comments on a first draft of this paper.

References

[Ban] Bela Ban. Design and Implementation of a Reliable Group Communication Toolkit for Java. http://www.cs.cornell.edu/home/bba/Coots.ps.gz.

[Ban98] Bela Ban. *JavaGroups User's Guide*. CS Dept. Cornell University, April 1998. http://www.cs.cornell.edu/home/bba/user/index.html.

[Bir96] Kenneth P. Birman. *Building Secure and Reliable Network Applications*. Manning Publications Co., 1996.

[Maf95] Silvano Maffeis. The Object Group Design Pattern. In *USENIX Conference on Object-Oriented Technologies (COOTS)*, June 1995.

[Sch86] Fred Schneider. The State Machine Approach: A Tutorial. Technical Report TR86-800, CS Dept Cornell University, 1986.

[Vay98] Alexey Vaysburd. *Building Reliable Interoperable Distributed Objects With The Maestro Tools*. PhD thesis, CS Dept Cornell University, May 1998.

[VRB95] Robbert Van Renesse and Kenneth P. Birman. Protocol Composition in Horus. Technical Report TR95-1505, Cornell University, March 1995. http://www.cs.cornell.edu/Info/Projects/HORUS/Papers.html.