GROUPZ Internals

José Pereira
jop@di.uminho.pt
U. Minho

May 11, 2011

1 Introduction

GROUPZ is a small and simple implementation of the virtually synchronous group communication abstraction [1] using the ZOOKEEPER distributed coordination service [3]. It provides view synchrony with sending view delivery and totally ordered multicast in a closed group [2]. The programming interface is minimalist and closely follows the naming and semantics of most literature on the topic. GROUPZ is open source and hosted at `http://gitorious.org/groupz`.

This document is a quick-and-dirty hackers guide to the implementation, structured as follows: Section 2 describes the system model assumed, which explains how we make use of ZOOKEEPER and expect a predictable correct outcome, even in the presence of local concurrency and exceptions. Section 3 delves into the details and should be read with an eye on the Java source code. Section 4 contrasts the abstractions offered by virtual synchrony and ZOOKEEPER. Finally, Section 5 lists some loose ends that need to be dealt with in the future.

2 System model

The system model assumed by GROUPZ is of a process group as a single distributed state-machine, encompassing all attached processes. The underlying state includes:

- Local variables, held locally by each process. These can be read and written only in transitions taken by the corresponding process.
- Global variables, held in shared memory within ZOOKEEPER. These can be read and written in transitions taken by any process.

There are also two different kinds of transitions:

- Input transitions are initiated outside GROUPZ and can happen at any time. Namely, these include sending a message, joining a group, agreeing to block, and leaving the group. If the input is not appropriate (e.g. sending a message while blocked), the transition is still done, although it might end in the terminal error state.
- Output transitions depend only on a pre-condition on state being true. The pre-condition is evaluated on all state variables, both local and shared. Output transitions include installing a view, deliveringa a message, and requesting to block.

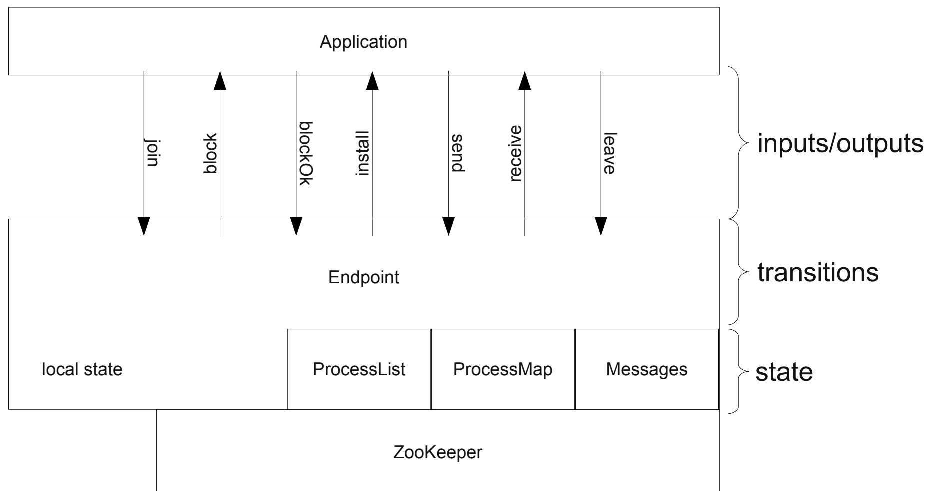

Figure 1: Architecture.

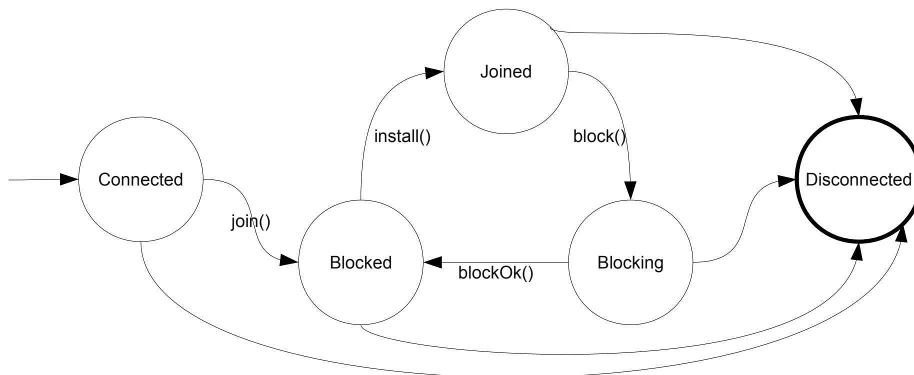

Figure 2: State machine.

As described below, thread synchronization, exception handling, and the agreement properties that ZOOKEEPER enforces on shared state mean that any run can be understood as a sequence of discrete transitions of a global state-machine. It is often convenient to consider the projection of this global state-machine on each process, including only its local variables and all shared state.

3 Implementation

The general architecture of GROUPZ is shown in Figure 1. Each process participates using an instance of class `Endpoint`. Local state is held in member variables and shared state is encapsulated in instances of `ProcessList`, `ProcessMap`, and `Messages`, also referenced by member variables. This allows code to observe and manipulate local and shared state uniformly.

Figure 2 shows the states and main transitions of the GROUPZ state-machine projected on each process. In short, when a process creates an `Endpoint` instance, it

becomes Connected. Upon joining, becomes Blocked waiting for a regular view change to complete. The first process to join the group, upon not finding any view to enter, creates an empty view 0, and then proceeds normally.

When Joined, the process will become Blocking when it discovers that another process wants to join or has left the group. After the application has agreed to block, the process becomes Blocked. This waits for all previously sent messages to become stable and then enter a new view, becoming Joined again.

While Joined or Blocked the process is allowed to send messages. While Joined, Blocked, or Blocking, the process is also allowed to receive messages. The process can always leave the group, volutarily or upon error, and become Disconnected.

3.1 Inputs

Input transitions are implemented as the external interface of class `Endpoint`. These transitions run in the calling application thread and are implemented as synchronized methods. The general template for an input action is:

```
  public synchronized ...(...) throws GroupException {
    onEntry(...);
    try {
      // code here
      ...
    } catch(Exception e) {
      onExit(e);
    }
}
```

It works as follows:

- An input action can only throw a `GroupException`.
- On entry, it verifies that the current state is allowed. Otherwise, it transitions to the Disconnected state and throws an exception.
- If any exception is thrown within, it transitions to the Disconnected state and throws an exception on exit.

3.2 Outputs

Output transitions run in an internal thread are visible by implementing the `Application` call-back interface. This thread is awaken whenever state varibales change in a way that might make a pre-condition true. When awake, it tests all pre-conditions and, if necessary, executes the corresponding transitions. In detail, this thread is awaken both directly by input methods (when changing local state) as well as indirectly by ZOOKEEPER "watchers", when shared state has changed.

```
  private boolean readyFor...() throws ... {
    return ...; // pre-condition here
  }

  private ...(...) throws ... {
    synchronized(this) {
```

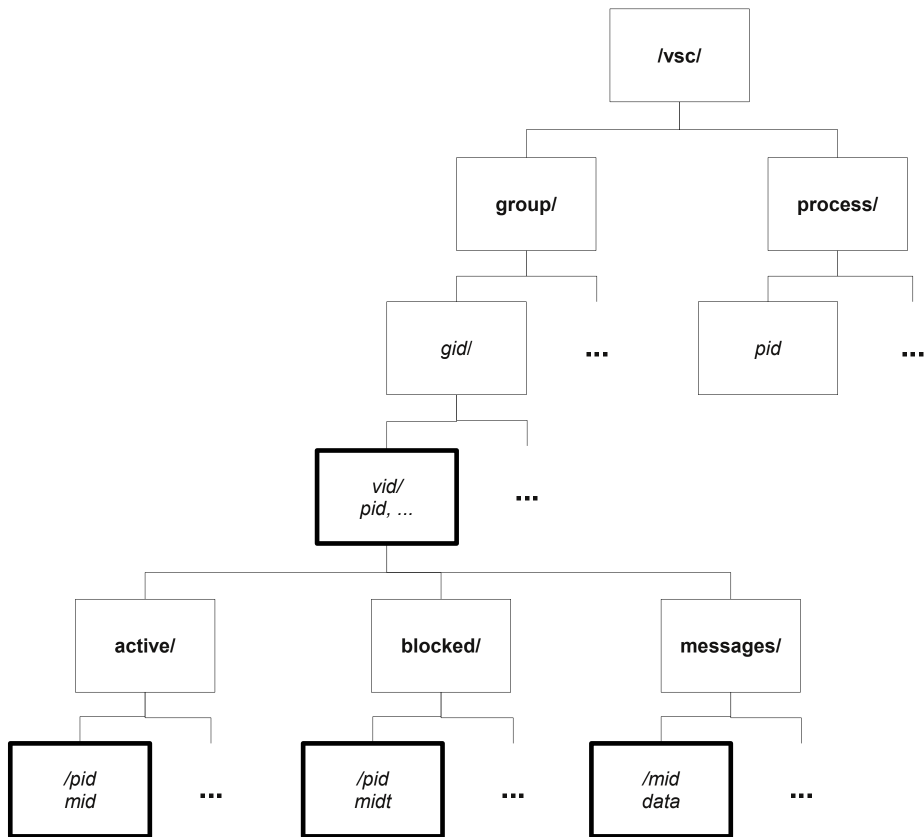

Figure 3: Shared state.

```
        if (!readyFor...()) return;

        // code here
        ...
    }
    ...(...); // call-back here
}
```

It works as follows:

- An output action can throw any exception. It will be caught by the main loop and force a transition to the Disconnected state. The cause is recorded and will be provided within `GroupException` if the `Endpoint` is used again.
- On entry, it verifies the pre-condition and give up if it is not satisfied.
- The application call-back must be invoked outside the synchronized block, to avoid that the application and output threads deadlock.

3.3 Shared state

Shared state kept in ZOOKEEPER is shown in Figure 3. All nodes are persistent unless noted otherwise. In detail:

- Each process gets its unique id by creating a sequential non-persistent node under /vsc/process/.
- Each groups stores all its data under /vsc/group/*gid* where *gid* is the symbolic name of the group.
- Each view is stored in /vsc/group/*gid*/*gid* where *vid* is the view number. This does not use ZOOKEEPER's sequential naming facility. As data, it contains the ordered list of process ids that are in the view and is observed and modified using the `ProcessList` class.
- Each view contains also a node messages. Sequentially named sub-nodes contain message data. Delivery must be ordered according to node names.
- Each view contains also nodes active and blocked. Non-persistent sub-nodes named with process ids and containing a message id hold the latest message received by each process. These data are observed and modified using the `ProcessMap` class. Moreover:
  - Processes in active are in the current view and willing to continue there. Whenever a process id disappears from this map, all others should initiate a view change.
  - Processes in blocked are willing to install a new view (whether they are in or out of the current view). Whenever a process id appears in this map, all others should initiate a view change. Stability during view change is computed in this map.

Each of the classes encapsulating shared state, namely `ProcessList`, `ProcessMap`, and `Messages`, registers as a ZOOKEEPER "watcher" and wakes up the thread in `Endpoint` to re-evaluate pre-conditions of output actions. Moreover, whenever queried, it refreshes its local copy of the state from the underlying ZOOKEEPER data, ensuring that a process that keeps querying the object will eventually get every update.

4 Discussion

The traditional virtually sycnhronous group communication model offers solutions to two distinct but equally hard distributed systems problems: Consensus and flow-control. First, consensus is implicit in agreement on view composition, on view ordering, and on totally ordered multicast. It is key to being able to maintain strongly consistent replicated state. Second, flow-control ensures that the system is paced according to (dinamically) available network and processing resources. Otherwise, congestion phenomena will greatly reduce throughput or even compromise the system's liveness.

ZOOKEEPER also provides distributed consensus, in particular, in ordering events leading to sequentially named data items and mutual exclusion when creating a path. Moreover, it provides also stable storage that persists across process crash and recovery faults.

By mapping these abstractions, GROUPZ is therefore restricted to the lowest common denominator: Consensus. It does not attempt to provide any form of distributed flow-control. On the other hand, it does not take advantage or try to expose stable storage to applications.

5 Work in progress

- Layering is still not perfect as `Endpoint` has to deal directly with a number of ZOOKEEPER paths. This should be fixed.
- Polling of enabled output actions can probably be reduced and made more efficient.
- The overall performance of this contraption remains a mistery, as nobody has ever bothered to measure it.

References

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

[2] G. Chockler, I. Keidar, and R. Vitenberg. Group communication specifications: a comprehensive study. *ACM Computing Surveys*, 33(4), December 2001.

[3] Flavio P. Junqueira and Benjamin C. Reed. The life and times of a zookeeper. In *SPAA '09: Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, pages 46–46. ACM, 2009.

Copyright and license

Copyright © 2010 José Orlando Pereira. Some rights reserved. Redistribution allowed according to Creative Commons Attribution-Share Alike 3.0 License. See `http://creativecommons.org/licenses/by-sa/3.0/` for details.