The Channel package

Throughout the book, we make use of a communication package to illustrate communication in distributed systems. The package provides a simple implementation of a bidirectional channel for passing messages between processes. In order for a process P_1 to send a message to another process P_2, both P_1 and P_2 will have to join the same channel. If multiple processes $P_1, \ldots, P_n$ have joined the same channel `c`, any process P_i can send a message to one other specific process P_j, or to every other process that has joined `c` (including itself). The package is written in Python.

The channel interface

The package provides a single class `Channel`, offering the following operations. Examples used below should clarify most of these operations.

`join(subgroup)` This operation is called to join a channel. The subgroup is a string indicating a possible subgroup of processes that the joining process belongs to. Typical examples of such sugbroups are "client," "server," "primary," and "backup." The operation returns a unique process identifier `pid` that can be used for communicating with the joined process.

`bind(pid)` Internally, each channel-level process is implemented as a separate operating-system-level process. This operation binds the channel-level process identifier to the process identifier as assigned by the operating system. As a consequence, when sending or receiving a message, the caller can be identified automatically.

`subgroup(subgroup)` Returns a list of process identifiers that have joined the channel in the subgroup named "subgroup."

`sendTo(destinationSet, message)` Called by a process to send the application-level message `message` to a set of destination processes, identified as a

list of process identifiers `destinationSet`. It is a nonblocking operation. Conceptually, `message` is copied to the list of input messages, with one list associated with each process. The message is prepended with the caller's identifier.

The calling process as well as each destination process is verified to have previously joined the channel.

`sendToAll(message)` To be called when the calling process wants the application-level message `message` to be sent to every process that has joined the channel (including itself). The message is prepended with the caller's identifier. The calling process is verified to have previously joined the channel.

`recvFromAny(timeout=0)` This operation returns to the calling process, the least recently message that was sent to it, prepended with the sender's identifier. That message is removed from the caller's input queue. If the input queue is empty, the caller is blocked until a message is sent, or until the specified `timeout` value (in seconds) has expired (a value of 0 means no timeout). Because each message is prepended with the sender's identifier, the calling process can always find out who sent the message. The calling process is verified to have previously joined the channel.

`recvFrom(senderSet, timeout=0)` This operation returns to the calling process, the least recently message that was sent to it by any process in `senderSet`. That message is removed from the caller's input queue. If there was no message from any process in `senderSet`, the caller is blocked until such a message is sent, or (only if `timeout` > 0) `timeout` seconds have expired. The calling process as well as the referenced senders are verified to have previously joined the channel.

Channels are implemented using the `Redis` package, which provides a server maintaining a (key,value) data store. By default, this data store is maintained on the local host, listening to default port 6379.

Some simple examples

To illustrate the usage of the package, consider the following simple examples. We confine ourselves to implementations running on a single machine, communicating with a single `redis` server listening to its default port.

A multiple client, single server system

We construct a simple client-server system by defining two classes, shown in Figure 0.1. The `Server` class initializes itself by joining a default channel

`channel.Channel()`, and subsequently joining the subgroup `server`. Once running, the only the server does is listen for incoming messages, and subsequently responding to the sender, acknowledging it received the message. Note that the sender is identified by `msg[0]`, and that a single-element list `[str(msg[0])]` is used for sending a response.

```
1  import channel
2
3  class Server:
4      def __init__(self):
5          self.ci=channel.Channel()
6          self.server=self.ci.join('server')
7
8      def run(self):
9          while True:
10             msg = self.ci.recvAny(self.server)
11             self.ci.sendTo(self.server, [str(msg[0])], 'Received '+msg[1])
12
13 class Client:
14     def __init__(self):
15         self.ci=channel.Channel()
16         self.client=self.ci.join('client')
17         self.server=self.ci.subgroup('server')
18
19     def run(self):
20         self.ci.sendTo(self.client,self.server,'Hello from '+self.client)
21         print self.ci.recvFrom(self.client,self.server[0])
```

Figure 0.1: Classes for client-server communication using channels.

The client is very similar. It joins the default channel, and, in particular, the subgroup `client`. In order to see who the server is, it asks for the processes from the `server` subgroup. Its lifecycle is simple: once run, it sends a "hello" message to the list of servers (in our example containing only a single process), to subsequently block until it receives a response.

To initiate a collection of 10 clients and a single server, the script shown in Figure 0.2 can be used. We assume that the client-server classes are contained in a file `clientserver.py`, whereas the channel package is available as a file `channel.py`.

We start with creating a channel `chan`, and by flushing its content we are sure that there are no entries left in the associated `redis` database from previous runs. A single server object and a total of 10 client objects are subsequently created. The actual server process is created through Python's `fork` operation and subsequently calling the server's `run` operation, effectively putting it into an infinite loop. Likewise, we then create the client processes by forking 10 more processes and subsequently calling `run` for each of them.

```
#!/usr/bin/env python
import os
import channel
import clientserver

chan = channel.Channel()
chan.channel.flushall()

server = clientserver.Server()
client = [clientserver.Client() for i in range(10)]

pid = os.fork()
if pid == 0:
    server.run()
    os._exit(0)

for i in range(10):
    pid = os.fork()
    if pid == 0:
        client[i].run()
        os._exit(0)
```

Figure 0.2: A script for starting a server and 10 clients as defined in Figure 0.1.