- Process resilience

The key approach to tolerating a faulty process is to organize several identical processes into a group. The key property that all groups have is that when a message is sent to the group itself, all members of the group receive it. In this way, if one process in a group fails, hopefully some other process can take over for it. Process groups may be dynamic. New groups can be created and old groups can be destroyed. A process can join a group or leave one during system operation. A process can be a member of several groups at the same time. Consequently, mechanisms are needed for managing groups and group membership.

Groups are roughly analogous to social organizations. Alice might be a member of a book club, a tennis club, and an environmental organization. On a particular day, she might receive mailings (messages) announcing a new birthday cake cookbook from the book club, the annual Mother's Day tennis tournament from the tennis club, and the start of a campaign to save the Southern groundhog from the environmental organization. At any moment, she is free to leave any or all of these groups, and possibly join other groups.

The purpose of introducing groups is to allow processes to deal with collections of processes as a single abstraction. Thus a process can send a message to a group of servers without having to know who they are or how many there are or where they are, which may change from one call to the next.

Flat Groups versus Hierarchical Groups

An important distinction between different groups has to do with their internal structure. In some groups, all the processes are equal. No one is the boss and all decisions are made collectively. In other groups, some kind of hierarchy exists. For example, one process is the coordinator and all the others are workers. In this model, when a request for work is generated, either by an external client or by one of the workers, it is sent to the coordinator. The coordinator then decides which worker is best suited to carry it out, and forwards it there. More complex hierarchies are also possible, of course. These communication patterns are illustrated in Fig. 8-3.

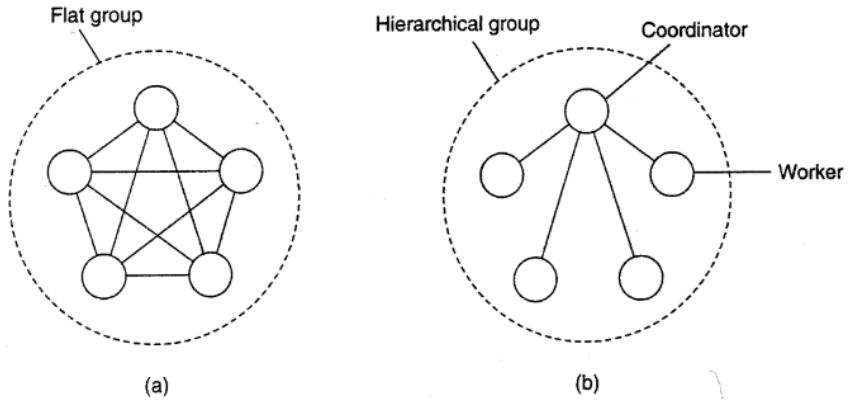

Figure 8-3. (a) Communication in a flat group. (b) Communication in a simple hierarchical group.

Each of these organizations has its own advantages and disadvantages. The flat group is symmetrical and has no single point of failure. If one of the processes crashes, the group simply becomes smaller, but can otherwise continue. A disadvantage is that decision making is more complicated. For example, to decide anything, a vote often has to be taken, incurring some delay and overhead.

The hierarchical group has the opposite properties. Loss of the coordinator brings the entire group to a grinding halt, but as long as it is running, it can make decisions without bothering everyone else.

Group Membership

When group communication is present, some method is needed for creating and deleting groups, as well as for allowing processes to join and leave groups. One possible approach is to have a group server to which all these requests can be sent. The group server can then maintain a complete database of all the groups and their exact membership. This method is straightforward, efficient, and fairly easy to implement. Unfortunately, it shares a major disadvantage with all centralized techniques: a single point of failure. If the group server crashes, group management ceases to exist. Probably most or all groups will have to be reconstructed from scratch, possibly terminating whatever work was going on.

The opposite approach is to manage group membership in a distributed way.

For example, if (reliable) multicasting is available, an outsider can send a message to all group members announcing its wish to join the group.

Ideally, to leave a group, a member just sends a goodbye message to everyone. In the context of fault tolerance, assuming fail-stop semantics is generally not appropriate. The trouble is, there is no polite announcement that a process crashes as there is when a process leaves voluntarily. The other members have to discover this experimentally by noticing that the crashed member no longer responds to anything. Once it is certain that the crashed member is really down (and not just slow), it can be removed from the group.

Another knotty issue is that leaving and joining have to be synchronous with data messages being sent. In other words, starting at the instant that a process has joined a group, it must receive all messages sent to that group. Similarly, as soon as a process has left a group, it must not receive any more messages from the group, and the other members must not receive any more messages from it. One way of making sure that a join or leave is integrated into the message stream at the right place is to convert this operation into a sequence of messages sent to the whole group.

One final issue relating to group membership is what to do if so many machines go down that the group can no longer function-at all. Some protocol is needed to rebuild the group. Invariably, some process will have to take the initiative to start the ball rolling, but what happens if two or three try at the same time?

The protocol must be able to withstand this.

Failure Masking and Replication

Process groups are part of the solution for building fault-tolerant systems. In particular, having a group of identical processes allows us to mask one or more faulty processes in that group. In other words, we can replicate processes and organize them into a group to replace a single (vulnerable) process with a (fault tolerant) group. As discussed in the previous chapter, there are two ways to approach such replication: by means of primary-based protocols, or through replicated-write protocols.

Primary-based replication in the case of fault tolerance generally appears in the form of a primary-backup protocol. In this case, a group of processes is organized in a hierarchical fashion in which a primary coordinates all write operations.

In practice, the primary is fixed, although its role can be taken over by one of the backups. if need be. In effect, when the primary crashes, the backups execute some election algorithm to choose a new primary.

An important issue with using process groups to tolerate faults is how much replication is needed. To simplify our discussion, let us consider only replicated write systems. A system is said to be k fault tolerant if it can survive faults in k components and still meet its specifications. If the components, say processes, fail silently, then having k + 1 of them is enough to provide k fault tolerance. If k of them simply stop, then the answer from the other one can be used.

On the other hand, if processes exhibit Byzantine failures, continuing to run when sick and sending out erroneous or random replies, a minimum of 2k + 1 processors are needed to achieve k fault tolerance. In the worst case, the k failing processes could accidentally (or even intentionally) generate the same reply.

However, the remaining k + 1 will also produce the same answer, so the client or voter can just believe the majority. Of course, in theory it is fine to say that a system is k fault tolerant and just let the k + I identical replies outvote the k identical replies, but in practice it is hard to imagine circumstances in which one can say with certainty that k processes can fail but k + 1 processes cannot fail. Thus even in a fault-tolerant system some kind of statistical analysis may be needed.

Failure Detection

It may have become clear from our discussions so far that in order to properly mask failures, we generally need to detect them as well. Failure detection is one of the cornerstones of fault tolerance in distributed systems. What it all boils down to is that for a group of processes, non faulty members should be able to decide who is still a member, and who is not. In other words, we need to be able to detect when a member has failed.

When it comes to detecting process failures, there are essentially only two mechanisms. Either processes actively send "are you alive?" messages to each other (for which they obviously expect an answer), or passively wait until messages come in from different processes. The latter approach makes sense only when it can be guaranteed that there is enough communication between processes.

In practice, actively pinging processes is usually followed.

There has been a huge body of theoretical work on failure detectors. What it all boils down to is that a timeout mechanism is used to check whether a process has failed. In real settings, there are two major problems with this approach. First, due to unreliable networks, simply stating that a process has failed because it does not return an answer to a ping message may be wrong. In other words, it is quite easy to generate false positives. If a false positive has the effect that a perfectly

a healthy process is removed from a membership list, then clearly we are doing something wrong.

Another serious problem is that timeouts are just plain crude. As noticed by Birman (2005), there is hardly any work on building proper failure detection subsystems that take more into account than only the lack of a reply to a single message. This statement is even more evident when looking at industry-deployed distributed systems.

There are various issues that need to be taken into account when designing a failure detection subsystem [see also Zhuang et al. (2005)]. For example, failure detection can take place through gossiping in which each node regularly announces to its neighbors that it is still up and running. As we mentioned, an alternative is to let nodes actively probe each other.

Failure detection can also be done as a side-effect of regularly exchanging information with neighbors, as is the case with gossip-based information dissemination (which we discussed in Chap. 4). This approach is essentially also adopted in Obduro (Vogels, 2003): processes periodically gossip about their service availability.

This information is gradually disseminated through the network by gossiping. Eventually, every process will know about every other process, but more importantly, will have enough information locally available to decide whether a process has failed or not. A member for which the availability information is old, will presumably have failed.

Another important issue is that a failure detection subsystem should ideally be able to distinguish network failures from node failures. One way of dealing with this problem is not to let a single node decide whether one of its neighbors has crashed. Instead, when noticing a timeout on a ping message, a node requests other neighbors to see whether they can reach the presumed failing node. Of course, positive information can also be shared: if a node is still alive, that information can be forwarded to other interested parties (who may be detecting a link failure to the suspected node).

This brings us to another key issue: when a member failure is detected, how should other non faulty processes be informed? One simple, and somewhat radical approach is the one followed in FUSE (Dunagan et al., 2004). In FUSE, processes can be joined in a group that spans a wide-area network. The group members create a spanning tree that is used for monitoring member failures. Members send ping messages to their neighbors. When a neighbor does not respond, the pinging node immediately switches to a state in which it will also no longer respond to pings from other nodes. By recursion, it is seen that a single node failure is rapidly promoted

to a group failure notification. FUSE does not suffer a lot from link failures for the simple reason that it relies on point-to-point TCP connections between group members.