**Remote Procedure Call**

This is a next level of using distributed systems, where we go beyond just sending data from one node to another. Form one node we send the message that calls the procedure on other node. While the second node does the procedure first node just waits for the answer or it can do something completely different to better use its time. After some time, second node sends the answer to first one.

Even though RPC is very useful, it has some problems. Frist of all, there are no global data structures, so remote process can only refer to received parameters. It creates the problem with linked lists, because they use pointers. Then we have problem with sending some complicated data structures and also the garbage collector that is local and cannot inform other node of his activities.

In using RPC, we assume that local procedure executes correctly. It is much higher probability that RPC will fail because of the network problems, server crash, client crash. In local procedures we have execute only once semantics. With RPC we cannot have that. We can have execute at least once (check account state, implemented like repeat after some timer), and execute only once (deposit and withdraw, server blocks repeat execution). First RPC implementation was made by SUN, and today we have Java RMI, and Corba

**Group Communication**

There are so many difficult things to consider when trying to make communication in distributed system. We need some middleware that will do all the complicated things for us. Then we could just call that functionality when we need it. We could have all or nothing semantics for broadcast sending, then solution for Byzantine agreement. We could for example want to send set of messages in some order and demand that processes receive it in that same order.

There are lots of terms that need to be defined for such communication. For starters, we need to define group, joining, leaving, crashing, end to end message sending, message broadcast to all group members, etc. There is also consensus problem that is not possible to solve in theory, but in practice we can use some sort of the trick to go around it. If we detect some traitor in the group, we kick him out of the group and try to make the consensus then (pharmacy, medicine and the assassin example). There is no defined set of services for this type of functionality even though it is part of every distributed system.

**Reliable Broadcast**

We need to broadcast message to all members of the group. If only one node doesn’t get it all other members drop the message.

In first implementation every node that receives the broadcasted message, broadcasts it again. That way if some node didn’t get the message he will surely get the copy. If it doesn’t get the copy it is just kicked out of the group because something is definitely wrong with it.

There is another approach that is similar to two faze commit protocol. We first broadcast prepare message, and wait for every node to send the acknowledge message. If some node doesn’t send it, we try again. And at the end we could kick it out of the group, and start without it, but with reliable broadcast.

There are much more like FIFO broadcast, Causal broadcast, Barrier, and more but we don’t go in to details here. Instead we jump to some more distributed algorithms.

**Replication**

We use this approach to mask crashes of the nodes in some distributed systems. In case of crash of some node we can always read from the last saved state in the system.

There are two strategies: **passive**, where we have one main node that does all the work. It periodically backs up necessary information to other nodes. In case of the crash it can recover using the last backup. **Active** where many nodes do the actual work, and if some of them crashes system can easily continue based on the work of the other nodes.