**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 by sending updates. In case of the crash it can recover using the last backup. Crash is detected if client doesn’t receive response from the server for some time. Then we can start a process of recovery using the replicas. There is also **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.

**Atomic Broadcast**

In replicated system, there is a problem when we need to coordinate multiple client requests on the many replicated servers. Handling request must be the same on all the replicas, so the order of request needs to be the same. As a solution we can use atomic broadcast.

Atomic broadcast is implemented in middleware layer where we have group communication over network layer. There are some properties that need to be implemented in group layer. If one process executes atomic broadcast, all the processes in his group will do atomic deliver. If at least one of the processes in the group do atomic deliver of some message, all the other processes in the group need to do the same. If two processes do atomic deliver of some two messages, they do it in the same order.

For the system with passive database replication, we can use generic broadcast instead of atomic one. It has a weaker form than atomic one in the sense that it doesn’t always enforce strict message ordering. If some messages need to have serial ordering, they can be marked as critical, and delivered in the right way. All the other ones that are sent concurrently, are not critical, and are sent freely. In this way we get grater performance for the systems with passive replication.

When implement atomic broadcast we have hard time at receiving end. There we need to remember set of messages on every node. We need to have consensus with all the other nodes in the group regarding that set of messages. If every node agrees on the set and ordering of its messages, we are free to deliver the messages.

**Virtual Synchrony**

True synchrony represents ideal view of the system where all the messages are delivered in exact moment they are sent. There we don’t have need to synchronize and order messages.

In the reality we have that messages need some time to be delivered and processed. Here we need the middleware to arrange and synchronize messages for the system to work correctly. It doesn’t try to develop total order where all the messages are delivered in the same order, but it just insists that system works correctly, so messages that need to be delivered in some order are delivered. All the other messages can be delivered in differed order for some users.

For its implementation, virtual synchrony needs three components: reliable broadcast with all-or-nothing semantics (use central server, or acknowledge), causal or totally ordered broadcast (use clocks), totally ordered membership updates.

Virtual synchrony has been used in Swiss stock exchange. Solutions are often limited in scale (50-70 members). If we have more users system gets slow.

**PAXOS**

PAXOS is a family of consensus protocols: cheap PAXOS, fast PAXOS, generalized PAXOS, byzantine PAXOS, and more. There are three roles present in PAXOS: one proposer, acceptors and learners. The whole point of it is that when more values are proposed, only one of proposed values will be chosen, only one is chosen, and only chosen values are learned. Depending on the problem we are trying to solve we will use one of many offered families in PAXOS.

First proposer sends prepare message with sequence number N. Acceptors accept it only if N is grater that last received N. If they accept, they send back last accepted value if they have it. Proposer must take the value that he received if at least one of acceptors already has accepted some value. If not, he can choose any value and send accept message. At the end if acceptors receive accept message with promised N, they accept the value and send it to learners.

If proposer crashes during the broadcast, new proposer will need to accept value from proposer that crashed, because one of the acceptors that accepted it will demand that.

// If I understood correctly

**Chubby** is lock service that works on the file system in distributed environment. It uses advisory, soft locks with notifications when someone changes files. Important property of Chubby is reliability, meaning that we can read and write despite of node crashes. Files that are used are very small, so more important is information in them. For example we can write in there who is the master currently, and every node can read easily who the master is.

At the end we don’t have in practice full or virtual synchrony. Google uses Chubby as very efficient naming system that has much weaker form of synchrony than real or virtual synchrony.