Network programming basics:

- One process listens for incoming connections.
- Other processes connect to the socket.
- Connection-oriented protocols (TCP) maintain the connection, and data can stream both ways.
- Connectionless protocols (UDP) send isolated packets and don't keep a connection open.
- TCP is like a telephone call, and UDP is like exchanging post cards or letters.
- Resolving, routing, and other communication details are abstracted away so the application programmer rarely needs to think about them.
- It's possible for the two communicating processes to be running on the same computer.
- The contents of the data stream (TCP) or packets (UDP) are not inherently structured— it's up to the application to decipher and encode them, just like files. This is known as the application-layer protocol, just as TCP and UDP are network-layer protocols and Ethernet is a data link-layer protocol. Examples: HTTP, FTP, NFS, IRC.

Traditional network programming:

- Each application imposes its own protocol on the otherwise-unstructured bits passed around on the network.
- Parsing and generating packets can be a tricky business due to:
  - Protocol definition documents (often called RFCs) can be vague about edge cases such as CR without LF.
  - Even if the RFC says "case is ignored," most implementors will only accept common capitalizations, and it is necessary to be compatible with these broken implementations.
  - Attackers can generate packets carefully designed to breach security— example: buffer overflow.
- Most protocols are very similar in structure: command word, argument list, CR LF, repeat. So a lot of work is duplicated when, say, writing separate HTTP and FTP servers.
- There are difficulties in escaping special characters (spaces, CRs, NULs), and length-prefixing isn't used nearly often enough.

Network programming with RPC:

- Each packet one process sends to another is for the purpose of requesting data, an action, or both, and the response packet can be synchronous, asynchronous, or omitted. This is generally true.
- This corresponds to making function calls within a single process. The function can accept arguments, may return a value, can block or trigger background processing, or may be in a separate thread.
- The idea is to standardize the wire protocol used by an entire class of applications, then have specific apps publish a procedure call interface. Now the same RPC protocol can be used for NFS or remote system administration or chatting.
- It's possible to create functions or objects in the local process that serve as proxies for functions or objects in the remote process— great for distributed systems!

RPC using the Tcl `comm` package:

- The traditional packet structure (command word, arguments, end-of-line) corresponds to the structure of a line of Tcl code.
- Additionally, Tcl has a means of encoding arbitrary data (structured or binary) as strings.
- The Tcl `comm` package marries RPC with Tcl script. Processes can send lines of Tcl code to other processes for execution, and the return value can be sent back synchronously or asynchronously.
- It's recommended to configure `comm` to accept a subset of Tcl that doesn't permit recursive command expansion or variable substitution. Also the range of allowed commands and acceptable quoting styles can be limited to those required for the specific application.
- This adds security by eliminating access to privileged commands, and it makes it possible to implement the application in languages other than Tcl, as the protocol has been reduced to a general RPC.