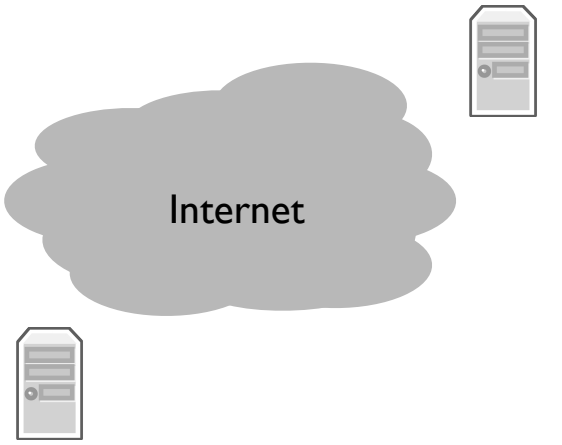
Networked Applications

What ultimately makes networks interesting are the applications that use them. Dave Clark, one of the key contributors to the Internet's design, once wrote "The current exponential growth of the network seems to show that connectivity is its own reward, and it is more valuable than any individual application such as mail or the World-Wide Web." Connectivity is the idea that two computers in different parts of the world can connect to one another and exchange data. If you connect your computer to the Internet, you suddenly can talk with all of the other computers connected on the Internet. Well, at least the ones that want to talk with you too. Let's look at what exactly that means and how some modern applications -- the world wide web, Skype, and BitTorrent -- use it.

Network Applications

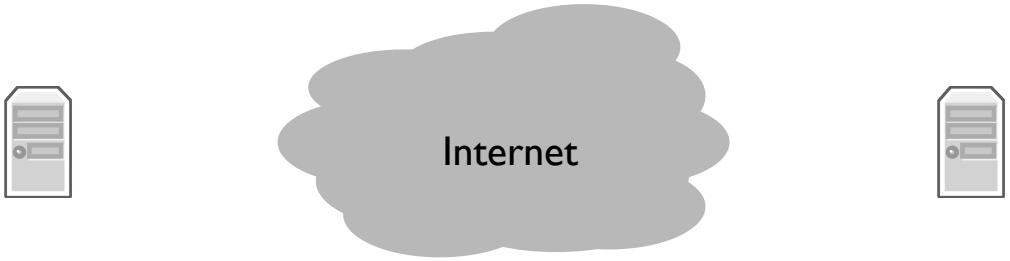


- Read and write data over network
- Dominant model: bidirectional, reliable byte stream connection
  - One side reads what the other writes
  - Operates in both directions
  - Reliable (unless connection breaks)

The tremendous power of networked applications is that you can have multiple computers, each with their own private data, each perhaps owned and controlled by different people, exchange information. Unlike your local applications, which can only access data that resides on your local system, networked applications can exchange data across the world. For example, think of using a web browser to read a magazine. The server run by the publisher has all of the magazine articles, and might also have all of the articles from past issues. As articles are corrected or added, you can immediately see the newer versions and newer content. The entire back catalog of articles might be too much for you to download, so you can load them on demand. If you didn't have a network, then you'd need someone to send you a DVD or USB stick with the latest issue.

So the basic model is that you have two computers, each running a program locally, and these two programs communicate over the network. The most common communication model used is a bidirectional, reliable stream of bytes. Program A running on computer A can write data, which goes over the network, such that then program B running on computer B can read it. Similarly, program B can write data that program A can read. There are other modes of communication, which we'll talk about later in the course, but a reliable, bidirectional byte stream is by far the most common one.

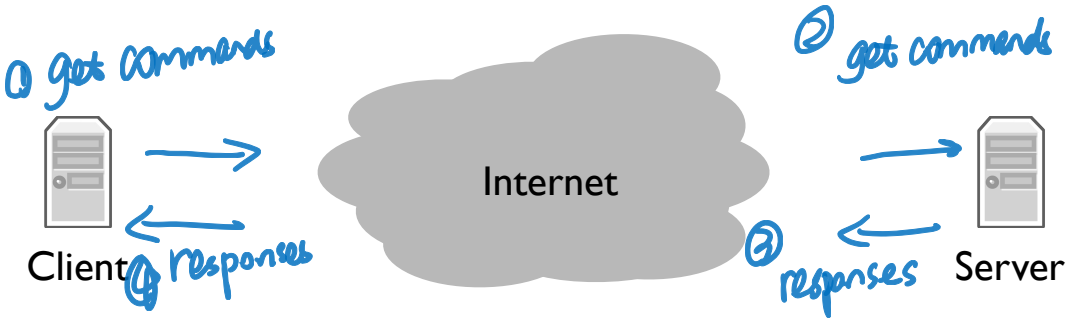
Byte Stream Model



Let's walk through what this looks like. Computer B, on the right, is waiting for other computers to connect to it. It might be, for example, a web server. Computer A, on the left, wants to communicate with B. Following this example, it's a mobile phone running a web browser. A and B set up a connection. Now, when A writes data to the connection, it travels over the network and B can read it. Similarly, if B writes data to the connection, that data travels over the network and A can read it. Either side can close the connection. For example, when the web browser is done requesting data from the web server, it can close the connection. Similarly, if the server wants to, it can close the connection as well. If you've ever seen an error message in a web browser saying "connection reset by peer," that's what this means: the web server closed the connection when the web browser wasn't expecting it. Of course the server can refuse the connection as well: you're probably seen connection refused messages, or have a browser wait for a long time because the server isn't even responding with a refusal.

Later in this course, you'll learn all of the details of how this works under the covers; for now, let's just think about it from the application perspective, which is the ability to reliably read and write data between two programs over a network.

World Wide Web (HTTP)

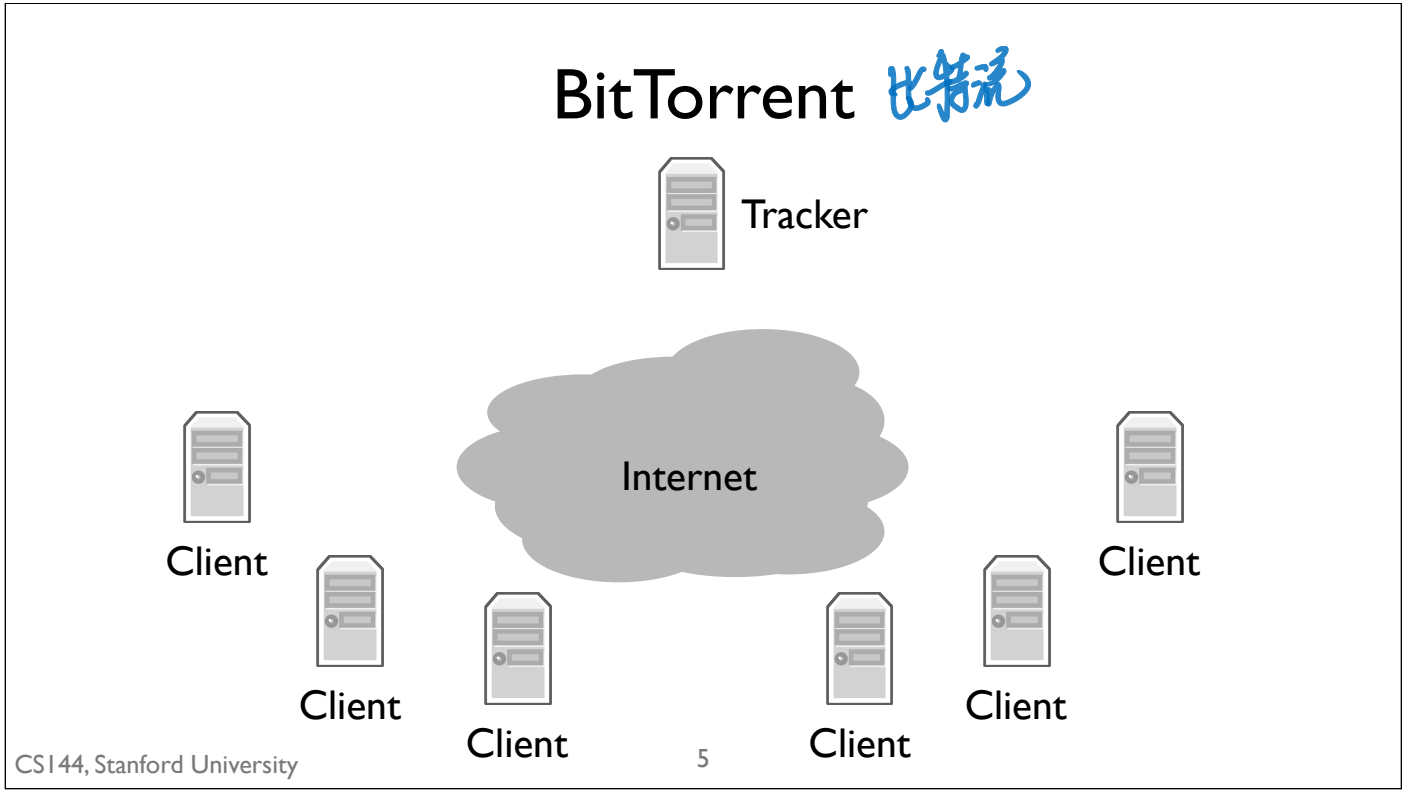


Now that we've seen the basic way networked applications communicate, let's look at our first example: the world wide web. The world wide web works using something called HTTP, which stands for the HyperText Transfer Protocol. When you see http:// in your browser, that means it's communicating using HTTP. We'll dig much deeper into the details of HTTP later in the course, when we cover applications. For now I'm just going to give a very high level overview.

In HTTP, a client opens a connection to a server and sends commands to it. The most common command is GET, which requests a page. HTTP was designed to be a document-centric way for programs to communicate. For example, if I type http://www.stanford.edu/ in my browser, the browser opens a connection to the server www.stanford.edu and sends a GET request for the root page of the site. The server receives the request, checks if it's valid and the user can access that page, and sends a response. The response has a numeric code associated with it. For example, if the server sends a 200 OK response to a GET, this means that the request was accepted and the rest of the response has the document data. In the example of the www.stanford.edu web page, a 200 OK response would include the HyperText that describes the main Stanford page. There are other kinds of requests, such as PUT, DELETE, and INFO, as well as other responses such as 400 Bad Request.

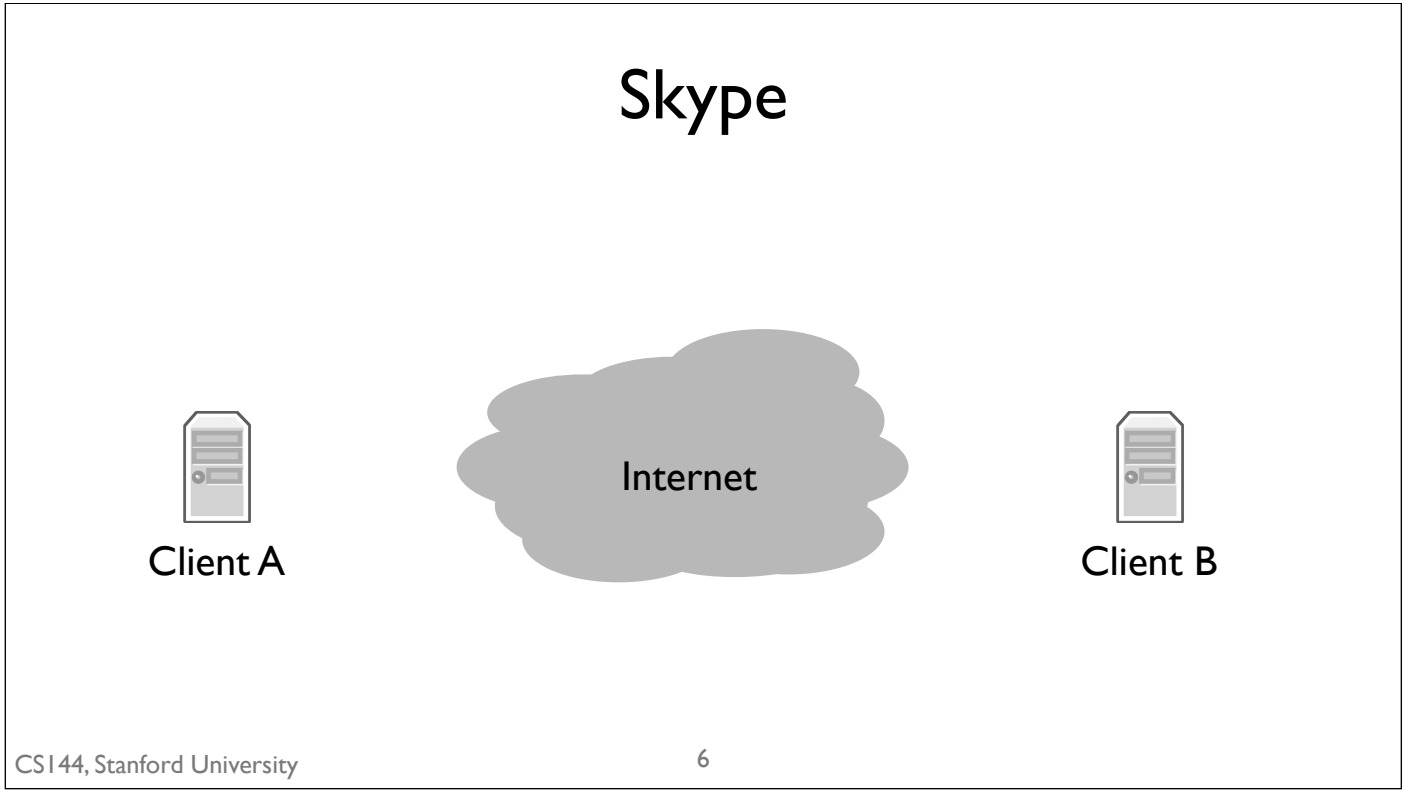
Because HTTP is document-centric, clients requests name a file. HTTP is all in ASCII text: it's human readable. For example, the beginning of a GET request for Stanford looks like this: GET / HTTP/1.1. The beginning of a response to a successful request looks like this: HTTP/1.1 200 OK.

But the basic model is simple: client sends a request by writing to the connection, the server reads the request, processes it, and writes a response to the connection, which the client then reads.



Let's look at a second application, BitTorrent. BitTorrent is a program that allows people to share and exchange large files. Unlike the web, where a client requests documents from a server, in BitTorrent a client requests documents from other clients. So that a single client can request from many others in parallel, BitTorrent breaks files up into chunks of data called pieces. When a client downloads a complete piece from another client, it then tells other clients it has that piece so they can download it too. These collections of collaborating clients are called swarms. So we talk about a client joining or leaving the swarm.

BitTorrent uses the exact same mechanism as the world wide web: a reliable, bidirectional data stream. But it uses it in a slightly more complex way. When a client wants to download a file, it first has to find something called a torrent file. Usually, you find this using the world wide web and download it using, you guessed it, HTTP. This torrent file describes some information about the data file you want to download. It also tells BitTorrent about who the tracker is for that torrent. A tracker is a node that keeps track (hence the name) of what clients are members of the swarm. To join a torrent, your client contacts the tracker, again, over HTTP, to request a list of other clients. Your client opens connections to some of these clients and starts requesting pieces of the file. Those clients, in turn, can request pieces. Furthermore, when a new client joins the swarm, it might tell this new client to connect to your client. So rather than a single connection between a client and one server, you have a dense graph of connections between clients, dynamically exchanging data.

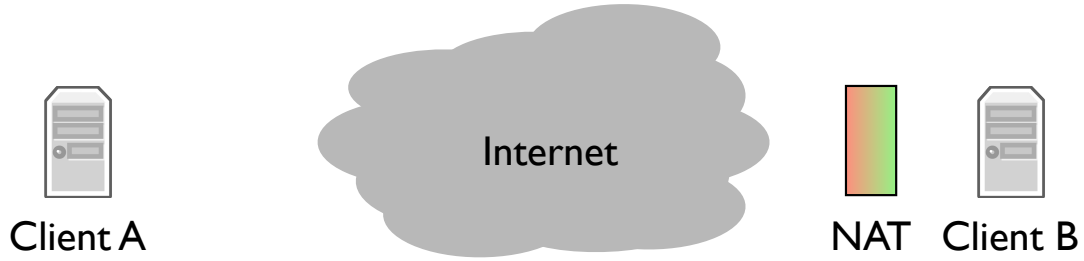


For our third and final application, let's look at Skype, the popular voice, chat, and video service. Skype is a proprietary system. It doesn't have any official documentation on how it works internally. In 2008 some researchers at Columbia figured out mostly how it works by looking at where and when Skype clients send messages. The messages were encrypted, though, so they couldn't look inside. In 2011, however, Efim Bushmanov reverse engineered the protocol and published open source code. So now we have a better sense of how the protocol works.

In its most simple mode, when you want to call someone on Skype, it's a simple client-server exchange, sort of like HTTP. You, the caller, open a connection to the recipient. If the recipient accepts your call, you start exchanging voice, video, or chat data.

In some ways this looks like the world wide web example: one side opens a connection to the other and they exchange data. But unlike the web, where there's a client and a server, in the Skype case you have two clients. So rather than having a personal computer request something from a dedicated server, you have two personal computers requesting data from each other. This difference turns out to have a really big implication to how Skype works.

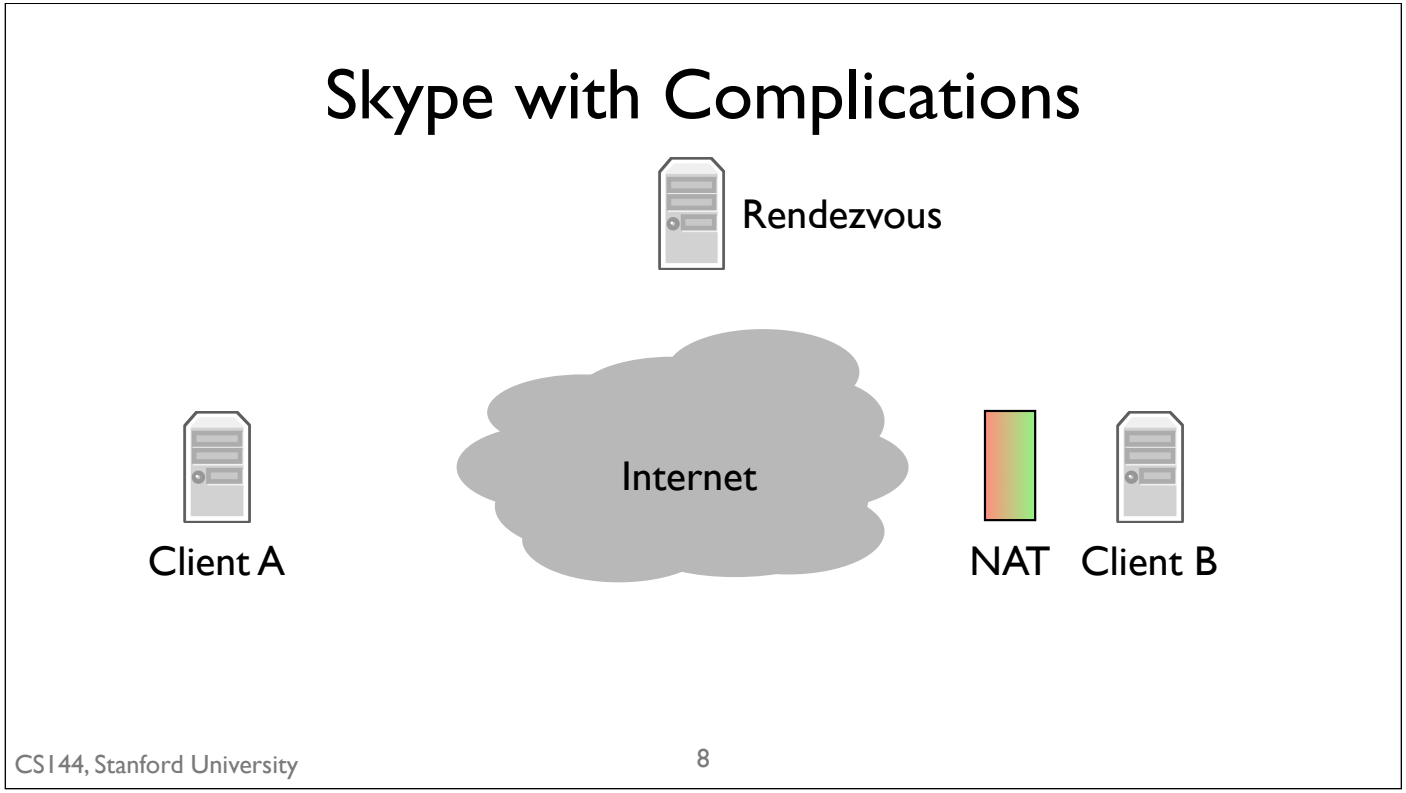
Skype with Complications



The complication comes from something called a NAT, or Network Address Translator. NATs are everywhere today. A small home wireless router is a NAT. When a mobile phone connects to the Internet, it's behind a NAT.

We'll cover them in greater detail later in the course, but for now all you need to know is that if you're behind a NAT then you can open connections out to the Internet, but other nodes on the Internet can't easily open connections to you. In this example, that means that Client B can open connections to other nodes freely, but it's very hard for other nodes to open connections. That's what this red-green gradient is showing; connections coming from the green side work fine, but connections coming from the red side don't.

So the complication here is that if the client A wants to call the client B, it can't open a connection. Skype has to work around this.

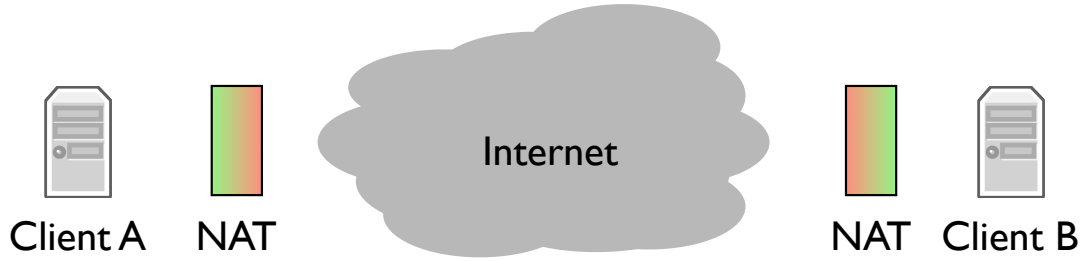


It does so using something called a rendezvous server. When you log into Skype, your client opens connections to a network of control servers. In this case, client B opens a connection to the rendezvous server. This works fine because the server isn't behind a NAT and client B can open connections out without any problems.

When client A calls client B, it sends a message to the rendezvous server. Since the server has an open connection to client B, it tells B that there's a call request from A. The call dialog pops up on client B. If client B accepts the call, then it opens a connection to client A. Client A was trying to open a connection to client B, but since B was behind a NAT, it couldn't. So instead it sends a message to a computer that client B is already connected to, which then asks client B to open a connection back to client A. Since client A isn't behind a NAT, this connection can open normally. This is called a reverse connection because it reverses the expected direction for initiating the connection. Client A is trying to connect to client B, but instead client B opens a connection to client A.

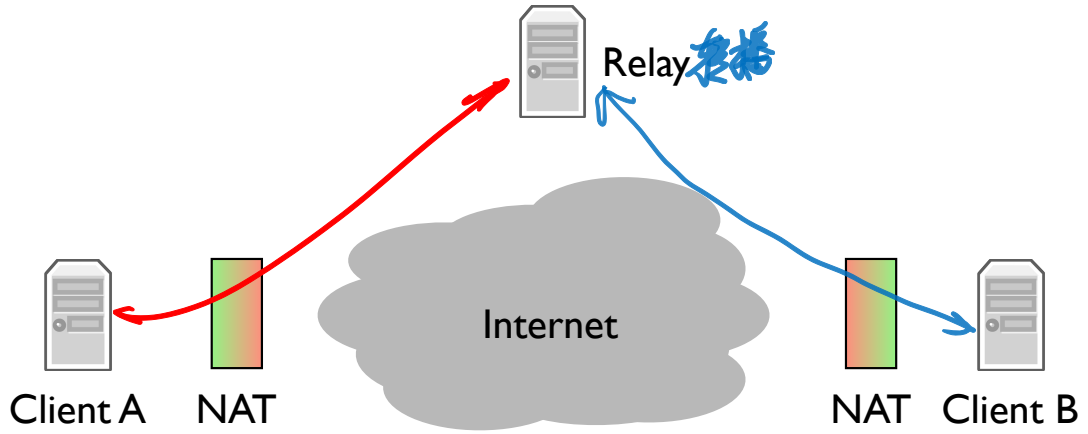
This happens in Skype because Skype clients are typically personal machines. It's rare for publicly accessible web servers to be behind NATs. Since you want the server to be accessed by everyone on the Internet, putting it behind a NAT is a bad idea. Therefore, opening connections to web servers is easy. Personal computers, however, are often behind NATs, for security and other reasons. Therefore Skype has to incorporate some new communication patterns to work around them.

Skype with More Complications



So what does Skype do if both clients are behind NATs? We can't reverse the connection. Client A can't open a connection to client B and client B can't open a connection to client A.

Skype with More Complications



To handle this case, Skype introduces a second kind of server, called a relay. Relays can't be behind NATs. If both client A and client B are behind NATs, then the communicate through a relay. They both open connections to the relay. When client A sends data, the relay forwards it to client B through the connection that B opened. Similarly, when client B sends data, the relay forwards it to client A through the connection client A opened.

Application Communication

- Bidirectional, reliable byte stream
  - Building block of most applications today
  - Other models exist and are used, we'll cover them later in the class
- Abstracts away entire network -- just a pipe between two programs
- Application level controls communication pattern and payloads
  - World Wide Web (HTTP)
  - Skype
  - BitTorrent

In summary, we've seen the most common communication model of networked applications: a reliable, bidirectional byte stream. This allows two programs running on different computers to exchange data. It abstracts away the entire network to a simple read/write relationship.

Although it's a very simple communication model, it can be used in very inventive and complex ways. We looked at 3 examples: the world wide web, BitTorrent and Skype. The world wide web is a client-server model. A client opens a connection to a server and requests documents. The server responds with the documents. BitTorrent is a peer-to-peer model, where swarms of clients open connections to each other to exchange pieces of data, creating a dense network of connections. Skype is a mix of the two. When Skype clients can communicate directly, they do so in a peer-to-peer fashion. But sometimes the clients can't open connections directly, and so instead go through rendezvous or relay servers.

So you can see how what looks like a very simple abstraction, a bidirectional, reliable data stream, can be used in many interesting ways. By changing how programs open connections and what different programs do, we can create complex applications ranging from document retrieval to swarming downloads to IP telephony. Trackers in BitTorrent, for example, have very different data and a very different role than the clients, just as Skype has relays and rendezvous servers in addition to clients.

But of course the Internet is continually evolving. Who knows what the next new amazing application will be? While it might have a different communication pattern than these three, it will almost certainly use a bidirectional, reliable byte stream in some new configuration.