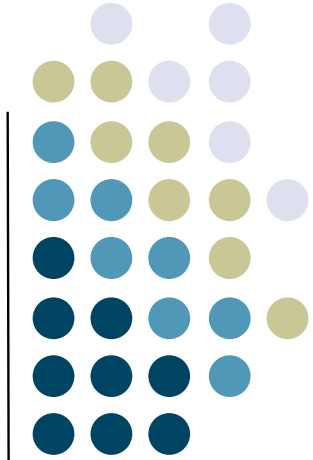
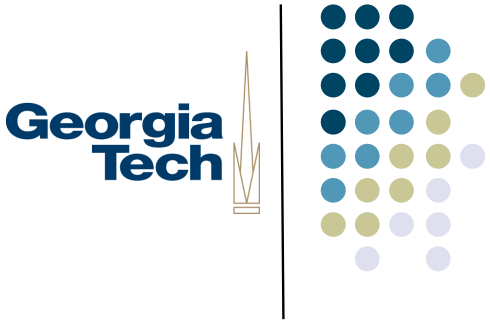
Peer-to-Peer Networking and Discovery Technologies



Georgia Tech

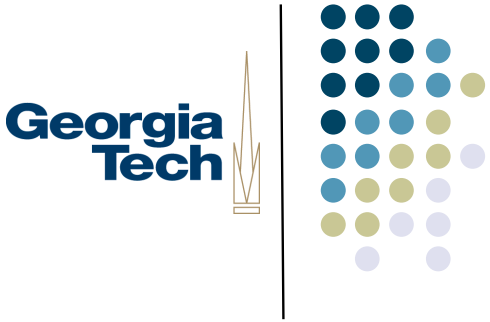
Week 6



What's Peer-to-Peer?

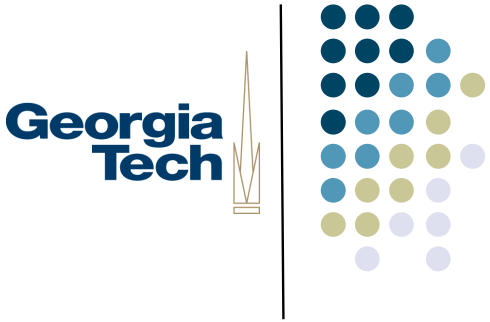
- A different network architecture than client/server
- Each peer is *both* a client *and* a server

- Appropriate for applications that don't require a centralized resource
- Really good for times when you may have connectivity to each other, but *not* connectivity to the Internet itself



Finding Resources

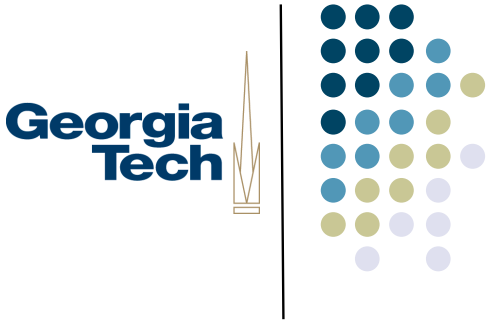
- Problem: how do you know what machines/services are available to you at any given time?
- Client/server: you have to *know* what you're looking for ahead of time, and then name it explicitly
  - E.g., typing google.com into a browser address field
- Peer-to-peer: more problematic
  - May not even know what's available to you
  - What's available may change rapidly
  - Hosts may not even have names (no DNS if you're off of the managed network, for instance)



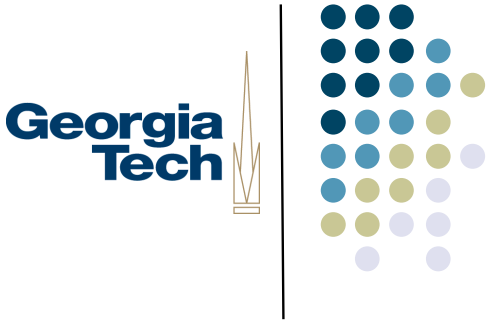
Solution: Discovery Protocols

- Discovery protocols are mechanisms for allowing a program to *dynamically acquire* references to resources it might want to use
  - Typically: IP addresses of available hosts, port numbers of available services, ...
- Address problems common in peer-to-peer (and other) networks:
  - Tell you what's available
  - Update this information based on current state of the network
  - Provide you with handles applications can use to access resources, even if they don't have human-readable names
- Process happens *automatically*: no end-user intervention

Why Discovery Protocols are Cool

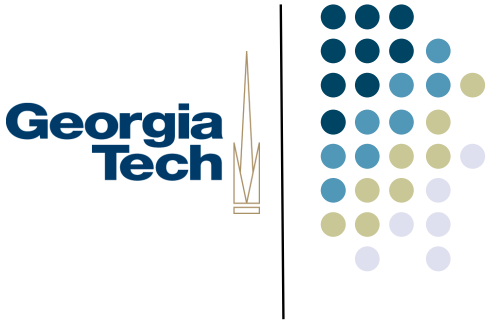


- Allow the creation of very dynamic applications
  - E.g., iTunes music sharing
- Allow you to get work done even if you're not on the broader Internet
  - E.g., file transfer, chat, etc., when you're not on a hot-spot
- Better end-user ease-of-use: no typing in IP addresses, host names, etc.
  - E.g., automatic printer setup, automatic discovery of multiple TiVos, ...



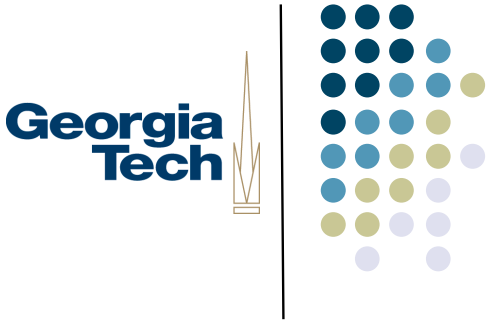
How They Work

- Lots of discovery protocols in use (Bluetooth, Universal Plug-and-Play, Zeroconf/Bonjour, ...)
- Lots of variation in details, but basics are generally the same:
  - Use a mechanism called *multicast*
  - Allows a program to send a single message that can be received by any number of other hosts
  - Typically configured to only work on a single network segment
  - Services/devices *announce* their presence using a periodic multicast message
  - Interested parties set themselves up to *listen* for these announcements



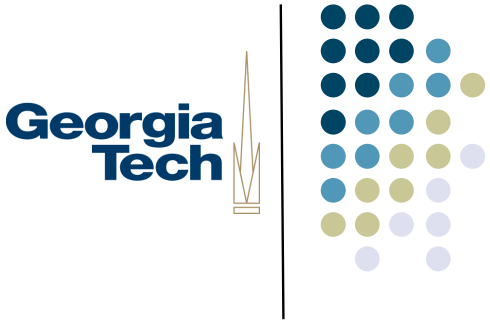
Example: Zeroconf

- Popularized by Apple
  - Also called Rendezvous, Bonjour, OpenTalk, ...
  - Basis for iTunes music sharing, photo sharing, iChat, ...
- Builds atop other Internet technologies
  - Provides a *multicast version of DNS* to allow name resolution to work in the absence of managed DNS servers
  - Extends DNS to allow *service discovery* information to be exchanged in DNS records
- mDNS: new domain: *.local.*
  - Names in this domain presumed to be meaningful only on the local link
  - Analogous to private/non-routed IP addresses
  - Attempt to resolve *tabasco.local.* triggers multicast to other computers, which can answer if they know the IP address of *tabasco*



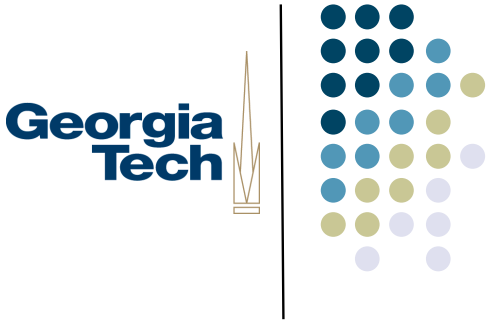
Example: Zeroconf (cont'd)

- DNS-SD: allows use of DNS for service discovery
  - Clients name the *service types* they wish to find
    - Format: *_type._protocol.domain*
    - Example: *_http._tcp.local.* would refer to all HTTP servers in the *.local.* domain
    - Types are just strings that name application protocols (_ftp, _http, _ssh, ...)
  - DNS-SD then returns a list of *service names* that match that type
    - Human-readable names that identify the service
    - Example: "Keith's web server on tabasco"
  - These service names are then resolved to an IP address and port number
    - Example: looking up *Keith's web server on tabasco* might return 192.168.1.177:80
    - This info can then be used to contact the web server



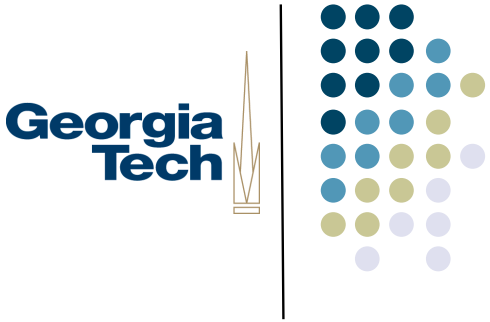
Using Zeroconf in Practice

- My recommendation: jmdns
  - Pure Java implementation of mDNS (and DNS-SD), callable from Jython
- Steps to using it:
  - Start your service, recording its port number
  - Publish the service
    - Choose a human-readable service name to publish under (your user name, for example)
    - Set the service type to be the name of the protocol we'll agree upon
      - _cs6452._tcp.local.
    - Publish its IP address and port number, which other apps will use order to connect to it
  - Register for notifications about peers
    - Tell jmdns that you want to know about services of type _cs6452._tcp.local.
    - Provide callbacks that jmdns will invoke when services come or go
    - When a service appears, you may need to ask jmdns to *resolve* that service's name, to get its IP address and port number
    - Once you have the address and port, you can connect to it using normal mechanisms



Gotchas, Tips, and Tricks

- Make sure your firewall is off, if you expect off-machine clients to be able to connect to you
- You can test and debug locally.
  - Publish different instances of your tool, under different names
  - Be sure to use different port numbers!
- Graphical mDNS browser:
  - java -jar lib/jmdns.jar -browse
  - Replace stuff in red with path to jmdns.jar



See Website for Examples

- jmdns-example.py
- Plus a jmdns "cheat sheet" with setup instructions