Washington State University

World Class. Face to Face.

# Distributed Programming (Ch. 10)

# Why Distributed

- Natural decomposition of the problem
  - Examples:
    - chat system – user agents and server;
    - storefront – customer agent and store server;
    - file sharing – peer-to-peer interactions between hosts that have files and hosts that want them
- Performance
  - Apply multiple processors to get the work done faster
- Availability
  - Backup resources to take over if primary fails
- Scalable problems – more resources for larger problems

# Erlang approaches

- "Distributed Erlang"
  - simple extension of the Erlang programming model to run on multiple computers (Erlang nodes)
  - Nodes are *authenticated*: to participate in a *cluster* a node must possess the cluster *cookie*
  - Nodes are *trusted:* fundamentally a node will do whatever another node in its cluster asks it to do
    - `rpc:call(bilbo@hauser-office, erlang, halt, []).`
- Socket-based distribution
  - Nodes have an *explicitly identified* set of services that they are willing to perform for other nodes

# Distributed Erlang

- What is a *node?*
  - A running instance of `erl`
    - Has a name given with -sname or -name switch
    - Has a hostname obtained from the machine on which it is running
    - A host may have many erlang nodes
- Primitives
  - `spawn(Node, Fun) spawn(Node, M, F, A)`
  - `spawn_link(Node, Fun) etc.`
  - Query functions for node identities, connectedness, etc. pp. 175-6

# Example

```
-module(echo).
-export([start/0, server/0]).

start() ->
    register(echo, spawn(fun() -> server() end)).

server() ->
    receive
       {Client, M} ->
            Client ! M,
            io:format("~p~n", [M])
    end,
    server()
    .
```

On another node

```
spawn('bilbo@hauser-desktop', echo, start, []).
{echo, 'bilbo@hauser-desktop'} ! "abc".
S2 = spawn('bilbo@hauser-desktop', echo, server, []).
S2 ! 17.
```

# Socket-based Distribution

- Node's available services defined in a configuration file
  `{port, Portnum}`
  `{service, S, password, P, mfa, Mod, Func, ArgsS}`
- `lib_chan:start_server(Conf)`
- `lib_chan:connect(Host, Port, S, P, ArgsC)`
  - Called on client
  - Returns `{ok, Pid}`
  - `Pid is a` *`local proxy`* `for talking to the server`
- When client connects call on the server:
  - `Mod:Func(MM, ArgsC, ArgsS)`

# Socket-based distribution (2)

- The server code has to be prepared to receive and send very specific messages – the socket distribution *protocol (see pp. 181-182)*
  - Client sends X to the proxy, server sees {chan, MM, X}
  - Server replies by sending {send, Result} to MM.
  - Server may see {chan_closed, MM} in mailbox meaning client disconnected
- Main thing to note: server only interacts with clients who know the password for that service *and* node only exposes specific service

# Security Concerns

- Distributed Erlang
  - "Cookie" knowledge gives node right to join group
  - Once in a group a node can do *anything at all* on other nodes
- Socket-based distribution
  - Per-service password (specified in config file)
  - All clients must know password – so all clients can disclose password

# Security Discussion

- Protocols are typically designed to *do something*
  - I.e., designed primarily to be obeyed
  - Security-specific protocols usually consider the behavior of attackers, but functionality-oriented protocols don't
- Protocol implementations
  - Work correctly on messages that obey the protocol
  - What do they do on messages that don't obey the protocol?
- Implementors' psychology? How should we as instructors teach students to approach implementation with security in mind?