Simulation of Skype Peer-to-peer Web Services Choreography Using Occam-Pi

Adrian Cockcroft while at eBay Research Labs

Paper Presented at IEEE EEE06/CEC06 June 2006

Author currently working at Netflix Inc.

acockcroft@netflix.com

Abstract

Complex web services are very difficult to test and verify before deployment on a large scale.

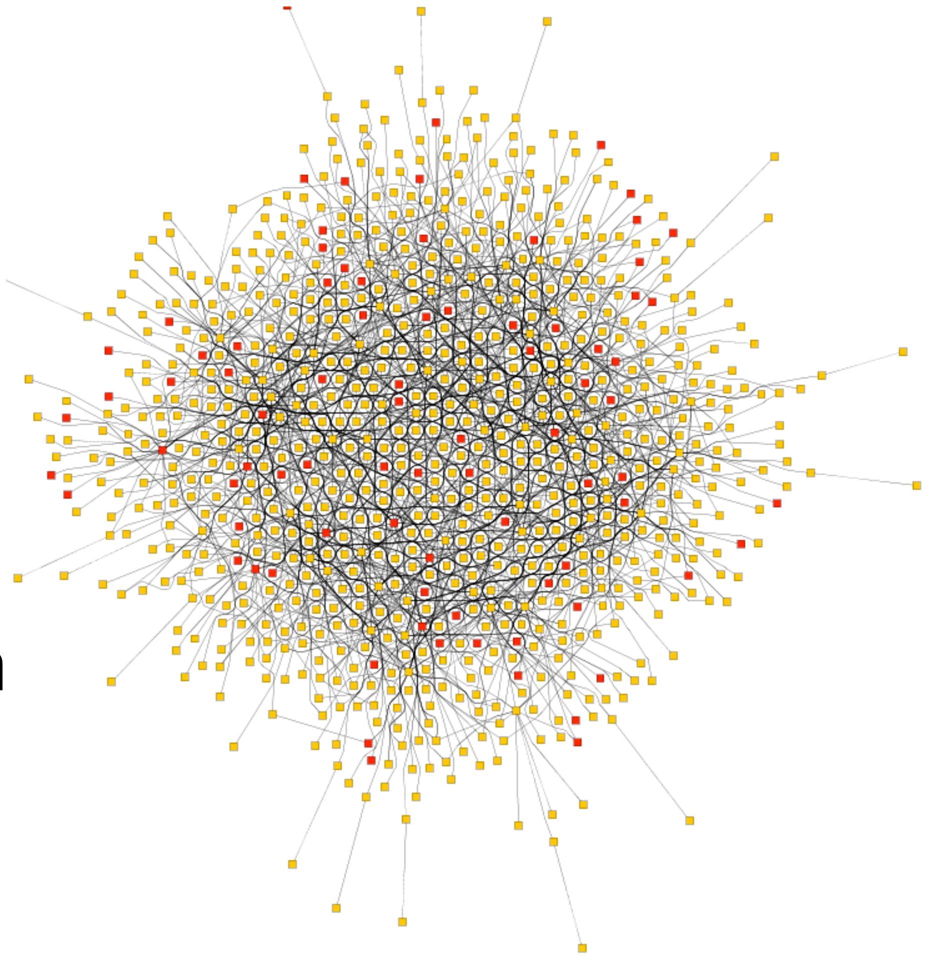
A semantically equivalent in-memory simulation can be built using Occam-pi that runs up to 100,000 nodes about a million times faster than real life.

Rapid protocol development and detailed behavioral analysis using simulation supports the development of robust services that can scale.

The implementation of a simulator that models centralized web services and application to application messaging over the Skype Peer to Peer network is described.

Overview

- The Landscape
- The Problem
- The Tools
- The Solution
- Implementation
- Running the Simulation
- Conclusion
- References



The Landscape – Simple Web Services

- Simple web services provide client access to centralized services in a star topology.
- A web client may integrate services from more than one central site to form a cluster of stars that are partially connected at the client nodes.
- The end result is mash-up's that provide access to information in better ways
- No need for simulation

The Landscape – Complex Web Services

- Complex web services choreography occurs when the topology is peer to peer, and any node provides and consumes services from any other node.
- The Business Process Execution Language [WS-BPEL] provides an orchestration standard. It helps compose higher level services from building blocks as a service state machine, tightly coupled towards an integrated whole as in an orchestral performance
- The Choreography Description Language [WS-CDL] describes how multiple parties can transact in a coordinated manner as a protocol state machine, dancing together without stepping on each others toes

The Landscape - Skype Peer to Peer Network

- What is Skype?
  - The worldwide leader in Voice over IP, and its free…
  - Over 550 Million downloads, over 100 Million registered users
  - Concurrent online population peaks at over 9 Million nodes
- What does Skype do?
  - Provide presence indications for when users are online
  - Provides high quality voice calls to and from Skype users and regular phones, voice mail and SMS messaging
  - Provides video-phone and instant message chat capability
  - Runs on Windows, MacOS, Linux and Mobile platforms
  - Has an open API for hardware and software integration
  - <u>Allows applications to talk over the peer to peer network!</u>

The Problem

- It is hard to design and verify large scale complex messaging systems, and the aggregate behavior of a choreography may exhibit undesirable behavior such as deadlock, livelock, node overload etc.
- Skype ap2ap API allows custom applications to define protocols to communicate directly over the Skype peer to peer network
- Existing applications mostly connect two peers, more interesting applications could connect many to many
- It is not practical to assemble large numbers of nodes for testing and development of a new protocol

The Tools – Pi Calculus

- WS-BPEL, SSDL and WS-CDL are based on Pi-Calculus
  - Pi-Calculus is based on Communicating Sequential Processes (CSP)
  - Pi-Calculus provides a formal model of parallel message based computing
- Occam-Pi
  - The Occam language is based on CSP, and has been extended to add the Pi-Calculus extensions to form the Occam-Pi language.
  - The primary implementation of this language is known as KROC, the Kent Re-targetable Occam Compiler
  - KROC is freely available from the University of Kent at Canterbury, UK
  - Runs on Intel architecture Linux, MacOS X, and Microsoft Windows/Cygwin platforms. Older versions exist for SPARC, PPC etc.

The Tools – Occam Language Constructs

- The constructs that are used in Web Services choreography map directly to Occam language constructs
- The Occam language has direct support for sequential, parallel and alternate processing blocks, complex protocols, and channel based communications
- Occam-Pi adds more dynamic constructs to the language
  - Mobile channels - pass a channel end over another channel
  - Mobile processes - suspend a process, pass it over a channel and resume it in a new context
  - Dynamic process forking with barrier synchronization
- Rigorous Occam-Pi Compile-time Checks
  - Processes or expressions are not allowed to have any “side effects”
  - Syntax and usage of all protocols, data and constructs is checked
  - Occam is designed to allow very comprehensive static analysis

The Tools – Occam Runtime Characteristics

- There is no need to use XML message encoding or namespaces
  - since the compiler can check that a protocol is being communicated correctly
- At runtime, the Occam-Pi language is fast small compiled code
  - with its own threading model, in a single process
- The Occam-Pi runtime detects and reports deadlock
  - including the line number in the code at which each process was stalled
- Occam-Pi is very efficient
  - All communication takes place in a single address space at memory bus speeds
  - Basic process creation takes 20 nanoseconds on an 800MHz PC
  - Basic channel communications takes 70 nanoseconds
  - Compared to typical web services transactions over the internet these transactions are about a million times faster
  - The language is also very compact, and one hundred thousand to one million threads can be created within a 2 GByte address space.

The Solution - Overview

- A simulator has been written in Occam-Pi
  - Implements a complex message exchange protocol
  - Models a hypothetical large scale Skype ap2ap deployment
  - Each simulated node contains its own private state and business logic and uses multiple threads
- Code size
  - The framework, command interpreter, logging and display is 1000 lines of code
  - Including the application specific protocol and business logic, 2000 lines of code
- The messages are specified as Occam-Pi Protocols
  - Reduced to their semantic content (enumerations, integers, strings etc.)
  - No XML tags, angle brackets or soap envelopes
- Benefits
  - Rapid development of protocols and messaging constructs
  - A large class of possible errors is caught at compile time
  - Easily extended to create the Java Message Classes or XML message formats

The Solution – Runtime Characteristics

- Network Size
  - For convenience and easy visualization,
    - a 1000 node network can be implemented
    - This results in a 25MB executing image while simulating a complex application
  - For large scale behavioral testing
    - with 100,000 nodes the process size grows to 360Mbytes
    - The simulation is usable with 100,000 nodes on a laptop with 1GB of RAM
- Log File Output from the simulation
  - Either dump everything in detail with microsecond resolution timestamps
  - Or produce a graph of the nodes and connections in GraphML format
  - GraphML is an XML dialect that can be read directly by tools such as yEd to visualize the network of connections
  - yEd can cope with 10000-100000 nodes and connections, gets slow….

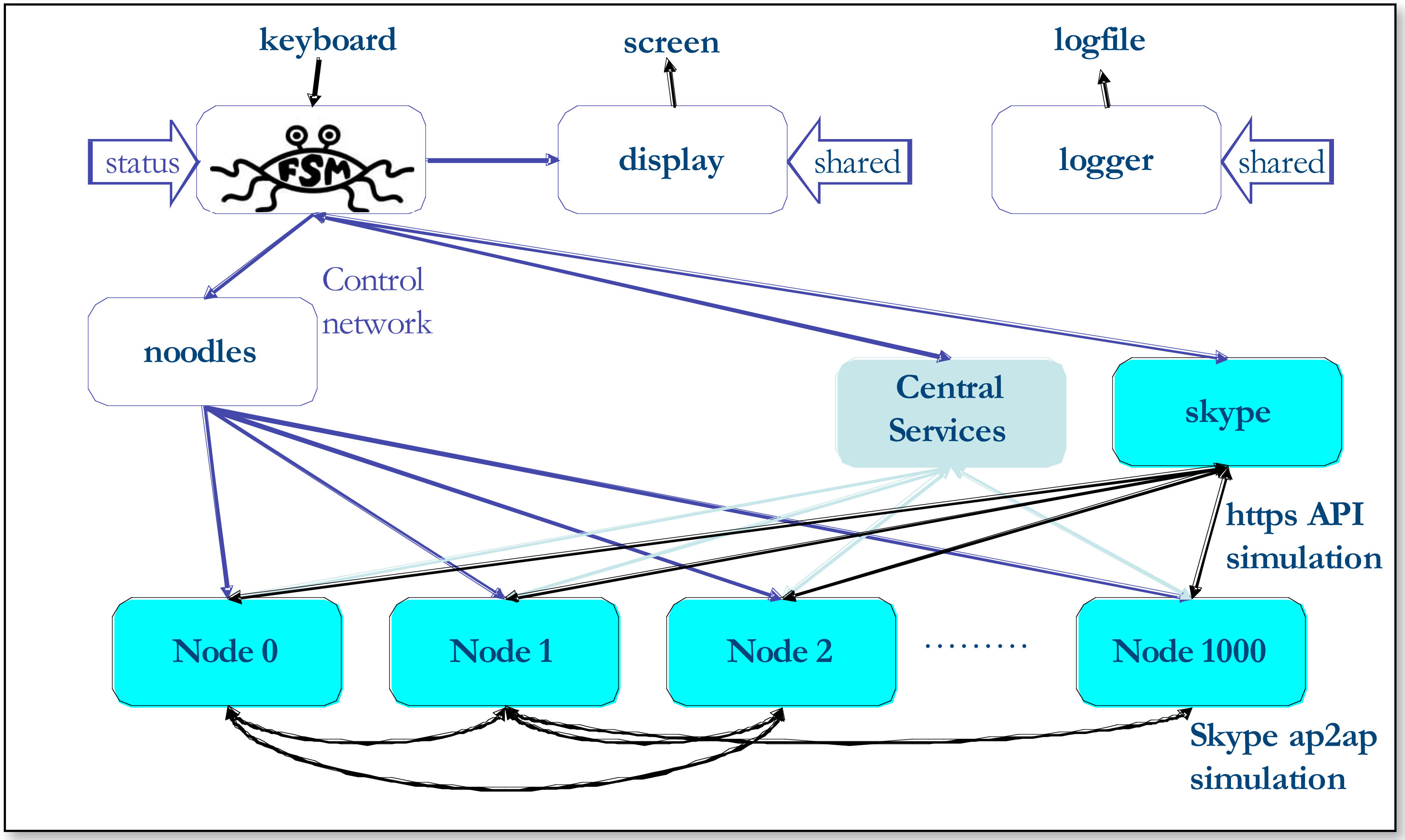
Implementation - Processes

- Simulation User Interface
  - Simple user interface uses a character mapped screen
  - There is no simple way to create a GUI directly from the Occam language
  - There exists a web server written in Occam, I didn't try to leverage it
  - Single character commands control aspects of the simulation
- Top Level Processes
  - Flexible Simulation Manager (FSM) that can see and control everything else
  - Screen display process
  - File logging process
  - Skype process that simulates the backend Skype services
  - Central services process implementing a custom backend for the application
  - Large number of node processes (typically 1000)
  - A “noodles” process that takes a single command from FSM and touches all the node processes in parallel to distribute the command while the FSM handles more user commands.

Implementation - Channels

- Simple point to point channels
  - Used for most connections
  - Hard coded in the top level process declarations
  - Marked with a ? sign if they are being used as input channels
  - Marked with a ! sign if they are being used as outputs
- Shared channels
  - Inputs to the FSM, Display, Skype, Logger
  - Node to Node channels
  - The sender blocks on a locking mechanism to claim the channel before they write a message
- Deadlock Prevention
  - Most outbound messages are implemented by forking a new thread that completes asynchronously
- Shared channels and forked threads are part of the recent Occam-Pi extensions to the language.

Implementation – Diagram



Implementation – Occam code for main process

```
--{{{ main PROC lambda
PROC lambda (CHAN BYTE kyb?, scr!)   -- connection to stdio of running process
  ...  declarations
  SEQ
    -- magically create many to one channels
    d.shared, d.chan := MOBILE MDISP
    api.put.shared, api.put.chan := MOBILE API.CALL.CHAN
    sc.shared, sc.chan := MOBILE SKYPE.CALL.CHAN
    chat.to.fsm.shared, chat.to.fsm.chan := MOBILE LISTEN.NODE.CHAN
    PAR
      display(d.chan?, scr!)
      fsm(kyb?, d.shared!, fsm.command!, chat.to.fsm.chan?, fsm.to.services!, fsm.to.skype!)
      noodles(fsm.command?, noodly.touch!)
      services(d.shared!, api.put.chan?, api.get!, fsm.to.services?)
      skype(d.shared!, sc.chan?, skype.response!, fsm.to.skype?)
      PAR i = 0 FOR num.nodes
        node (i, sc.shared!, skype.response[i]?, api.put.shared!, api.get[i]?,
         noodly.touch[i]?, chat.to.fsm.shared!)
:
--}}}
```

Implementation - Instant Tutorial Guide to Occam

- Occam code is structured using <u>indentation as part of the syntax</u>
- All reserved words and types are in UPPER CASE
- Occam is usually written using a folding editor such as vim
  - blocks of code are folded away to follow the structure more easily
  - –{{{ and --}}} markers show an open fold
  - The … marker shows a closed fold that hides a block of declarations.
- SEQ keyword is used to create a sequential execution block
  - First initialize some shared channels
  - In Occam-Pi, a channel that is shared at only one end is of a different type at each end
  - Some magical syntax dynamically binds the two ends of the channel
    - MOBILE keyword allocates a channel of a specified protocol and returns two channel ends
    - A list of comma separated variables can be assigned on the left side of the expression
- PAR keyword introduces a parallel execution block
  - Each line in this block is run in a separate parallel thread
  - The last line is a replicated PAR which creates 1000 parallel threads to run the nodes
  - When all elements of a PAR have completed, the block ends
    - In this case there are no more lines of code and this is the main program so it exits

Implementation – Protocol definition for Skype Backend

```
-- shared writer, single reader
CHAN TYPE P2PLINK
  MOBILE RECORD
    CHAN SKYPE.P2P c?:
:
-- shared incoming command, channel index used to route response
PROTOCOL SKYPE.CALL
  CASE
    register; INT; [8]BYTE             -- channel; skype name
    authenticate; INT; INT             -- channel; skype id
    find; INT; [8]BYTE                 -- channel; find someone with this name
   connect; INT; INT; SHARED P2PLINK!  -- fromid; toid; open connection
:
-- sharable chan type splits in and out ends
CHAN TYPE SKYPE.CALL.CHAN
  MOBILE RECORD
    CHAN SKYPE.CALL c?:
:
-- non-shared response channel array
PROTOCOL SKYPE.RESPONSE
  CASE
    -- skypid; skypename respond to register, auth and find messages
    registered; INT; [8]BYTE
    unregistered; [8]BYTE             -- bad name - can't register or find
    denied; INT                       -- bad id - can't authenticate id
    offline; INT                      -- can't connect to valid id
    connected; INT; [8]BYTE; SHARED P2PLINK! -- id, name and channel endpoint
:
```

Implementation - Skype Related Protocol Explanation

- P2PLINK is the protocol used between the nodes
  - SKYPE.P2P represents the application level protocol that takes place over the Skype ap2ap mechanism
  - I will not be discussing any details of the application that we simulated
  - Since P2PLINK will be created as a shared writer and single reader, it needs to be defined as a MOBILE RECORD
  - The colon terminates the definition.
- SKYPE.CALL defines the calls that can be made to the central Skype process
  - Uses a variant protocol, introduced by the CASE keyword
  - Each protocol variant starts with an enumerated tag, followed by the sequence of types that make up that protocol
  - The first protocol element in each case is an INT that identifies the node that is calling on Skype
  - The "register" variant is used to register the Skype name of a node, a fixed size 8 character array is used to hold the name
    - A unique numeric id is issued by the Skype process for each name it registers
  - The "authenticate" variant is used to authenticate a numeric Skype id
  - The "find" variant looks for a name to see if it is valid
  - The "connect" variant passes a shared channel end from one node to Skype
    - Skype passes it on to another node so that two nodes can be connected directly.
- SKYPE.CALL.CHAN implements a sharable channel type based on SKYPE.CALL
  - The central Skype process uses this to provide a single input channel for all nodes to make requests
- SKYPE.RESPONSE is the protocol used to respond to nodes
  - It implements simple responses that indicate whether a Skype id or name exists and can be connected to, and also implements the outbound “connected” protocol that passes a P2PLINK to its destination node.

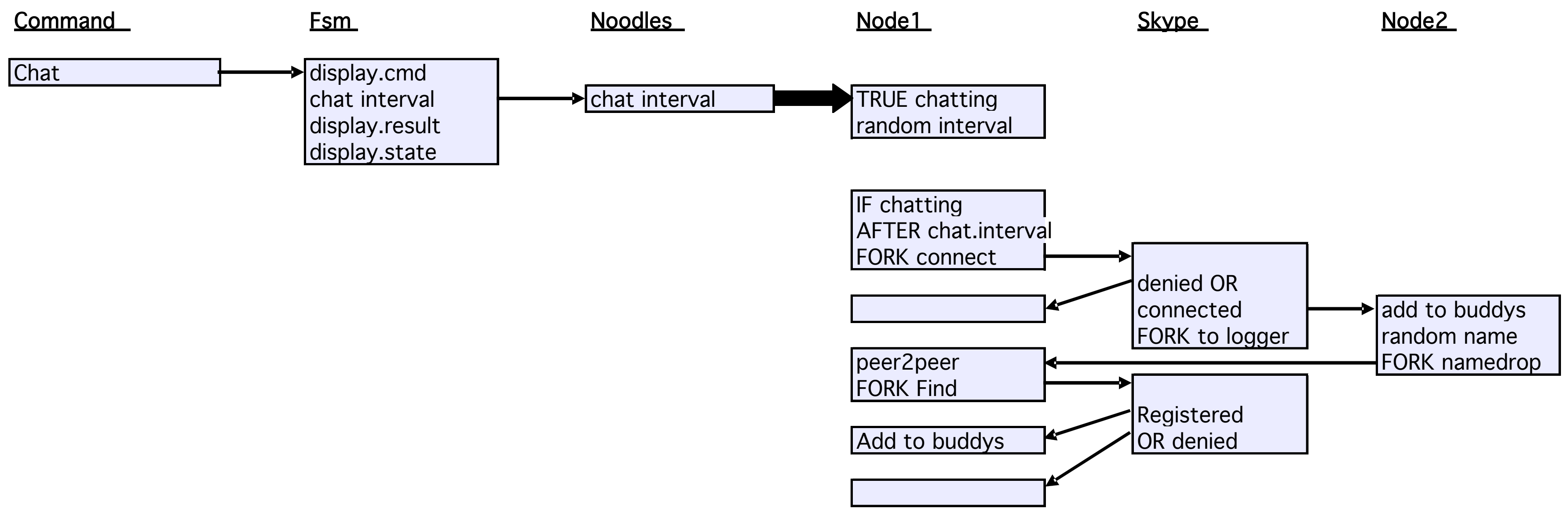
Implementation - Central Skype service process

```
PROC skype (SHARED MDISP! screen, SKYPE.CALL.CHAN? from.node, []CHAN SKYPE.RESPONSE to.node!, CHAN FSM.WORD from.fsm?)
  ...  declarations
  FORKING
    SEQ
      ...  initialization
      WHILE running
        PRI ALT
          from.fsm ? CASE
            ...  obey divine commandments
          awake & from.node[c] ? CASE
            --{{{ provide skype services
            register; route; name
              ...  register name and return id key in registered message
            authenticate; route; id
              ...  authenticate id key, return denied or registered
            find; route; name
              --{{{ look for a user name return registered or unregistered
              SEQ
                find.user(id, name, user.name, user.registered)
                IF
                  id = (-1)
                    to.node[route] ! unregistered; name
                  TRUE
                    to.node[route] ! registered; id; name
              --}}}
            connect; fromid; toid; listener     -- node request contains its listener channel
              --{{{ connect two nodes
              SEQ
                IF
                  (toid >= num.nodes) OR (toid < 0)
                    SEQ
                      ...  log denied
                      to.node[fromid] ! denied; toid
                  user.registered[toid]
                    SEQ
                      -- tell other node how to connect
                      IF
                        DEFINED logchan
                          SEQ
                            IF
                              GraphML
                                sprintf.lss(sh, len, "    <edge source=*"%s*" target=*"%s*"/>*n",
                                  user.name[fromid], user.name[toid])
                              TRUE
                                sprintf.lss(sh, len, "connected from %s to %s",
                                  user.name[fromid], user.name[toid])
                            mess := MOBILE [len]BYTE
                            mess := [sh FOR len]
                            FORK log.buffer.write(logchan!, "skype", mess)
                        TRUE
                          SKIP
                      to.node[toid] ! connected; fromid; user.name[fromid]; listener
                      peers := peers + 1
                  TRUE
                    SEQ
                      ...  log unregistered denied
                      to.node[fromid] ! denied; toid
               --}}}
            --}}}
:
```

Implementation - code for Central Skype Process Explained

- The outermost block in the code is a FORKING block
  - All subprocesses have to exit before the FORKING block completes, this acts as a barrier synchronization
  - The SEQ block that follows initializes then loops until the running boolean variable is cleared by a command from the FSM
- The process is structured as a prioritized alternate input
  - PRI ALT listens on multiple channels, and acts on whichever one is ready, giving priority to the first one
  - This is structurally similar to a select call on a socket, but is a fundamental Occam language construct
- The first channel is from.fsm? (the . is not significant in Occam names)
  - it implements a variant protocol that is used to control, monitor and terminate the simulation
- The second channel is awake & from.node[c]? and is a guarded channel record
  - The boolean guard awake is controlled by one of the FSM commands to enable/disable input on this channel
  - The subscript [c] is a record selector that is redundant syntax in this use of the construct
  - The variant protocol tags are specified in a CASE construct, i.e. register, authenticate, find and connect
  - With each tag are the local variables which will be instantiated with the data transferred
  - The find handler calls a routine that looks for the existence of a name then uses an IF conditional that matches each test in turn and sends a return message as appropriate
  - The connect handler first checks that the destination node id is in a valid range, then looks up the user in a boolean array, logs the connection, and sends a connected message to that node containing the listener channel of the requesting node
  - The logging code is conditional on the existence of a shared logging channel
  - The TRUE/SKIP construct is equivalent to a default empty “else” clause in other languages
  - If GraphML is required, then the node information is formatted for output as an XML string
  - The output buffer is allocated as a MOBILE array, like a malloc, and the string is copied as a slice using the [sh FOR len] syntax.

Running The Simulation – Swim Lane Diagram



Running the Simulation - Swim Lane Diagram Explained

- Simple example - "name drop"
  - At random intervals of a few seconds each node picks a buddy
  - Tries to connect to them by sending a message via Skype
  - The buddy node responds by picking the name of one of its own buddies at random and responding directly to the first node with that name
- Simulates the effect of namedropping a third party in a chat session
- Causes the nodes to propagate their connections and become more connected as they add to their buddy lists
- While running in this mode the CPU load is a few percent with 1000 nodes, and about 50% with 100,000 nodes measured on a Dell 1.6GHz Pentium-M laptop running Windows XP and Cygwin.

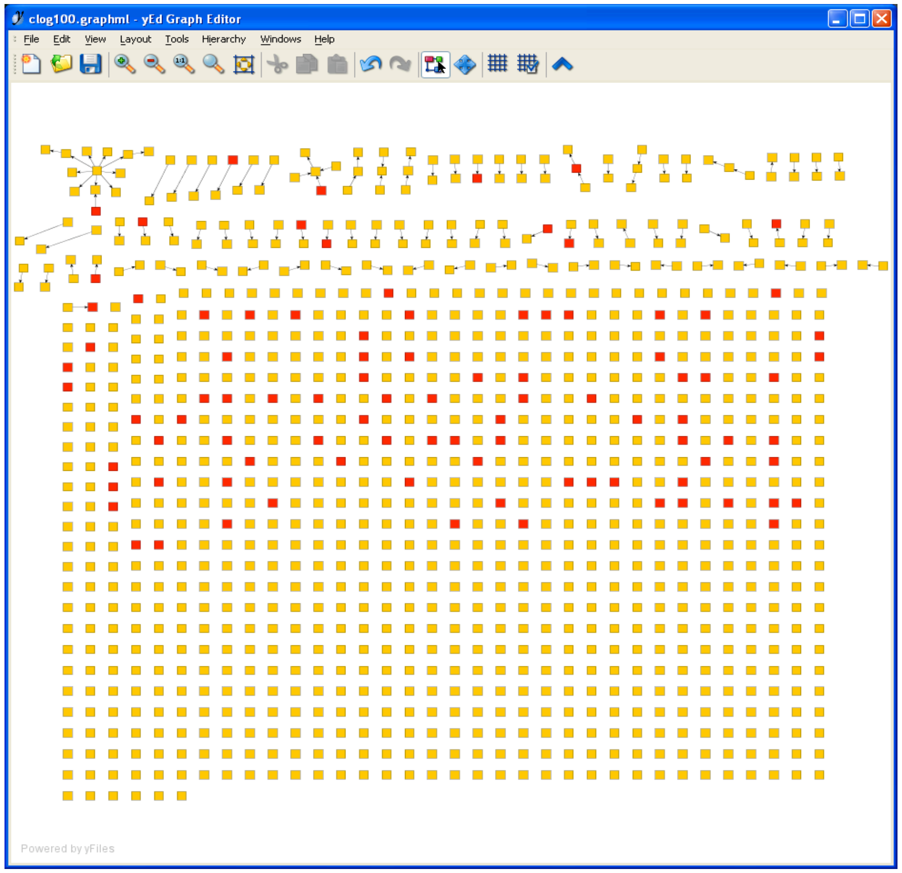
Running the Simulation - GraphML logfile format

```
<?xml version="1.0" encoding="UTF-8"?>
 <graphml xmlns="http://graphml.graphdrawing.org/xmlns/graphml"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://graphml.graphdrawing.org/xmlns/graphml
   http://www.yworks.com/xml/schema/graphml/1.0/ygraphml.xsd"
   xmlns:y="http://www.yworks.com/xml/graphml">
    <key id="d0" for="node" yfiles.type="nodegraphics"/>
    <key id="d1" for="edge" yfiles.type="edgegraphics"/>
    <graph id="LambdaNodes" edgedefault="directed">
     <node id="n0"/>
     ...
     <node id="n999"/>
     <edge id="e149" source="n211" target="n225"/>
     ...
   </graph>
 </graphml>
```

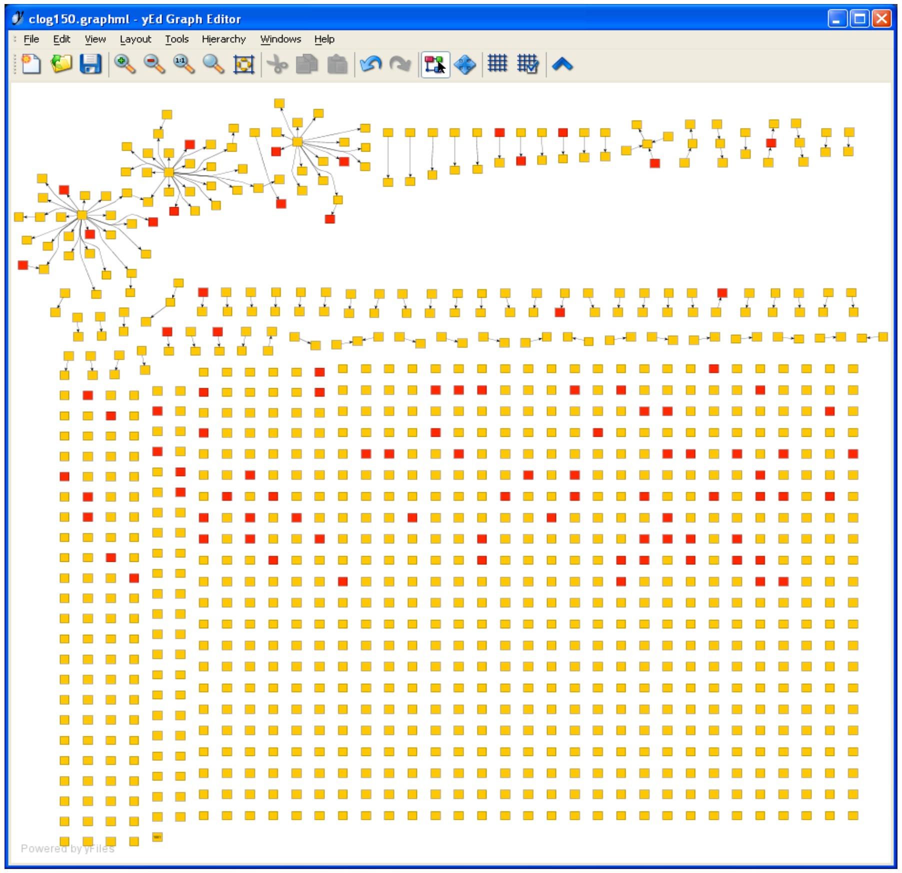
Visualization

- Simulation Results
  - When a simulation runs, the edges are added to the logfile in order
  - To illustrate the effect the simulator was run for a few seconds with 1000 nodes
  - Nodes chatting and namedropping created a few thousand connections.
  - Nodes have randomized wakeup intervals so the CPU load was a few percent for this run
- The most interesting period is the initial formation of the network
  - The log file was edited down and many versions were produced that incrementally show more connections
  - The resulting graphs are shown in subsequent slides
- yEd Graph Editor
  - layout used was “Organic – smart” with “organic edge routing”
  - There are many other possible display types
  - Visualization performance was acceptable with 1000 nodes and 1000 or so edges, but larger and more highly connected networks become harder to visualize effectively
  - Some of the nodes were colored red in this display, but this particular run did not depend on that attribute so they are not treated specially. Color can be used to visualize node states.
  - Edges could also be colored to indicate the protocol element in use
- Nodes wake up at intervals to communicate
  - The most active nodes form the initial network “hubs”
  - This behavior can be easily modified at will to simulate whatever scenario is deemed interesting.

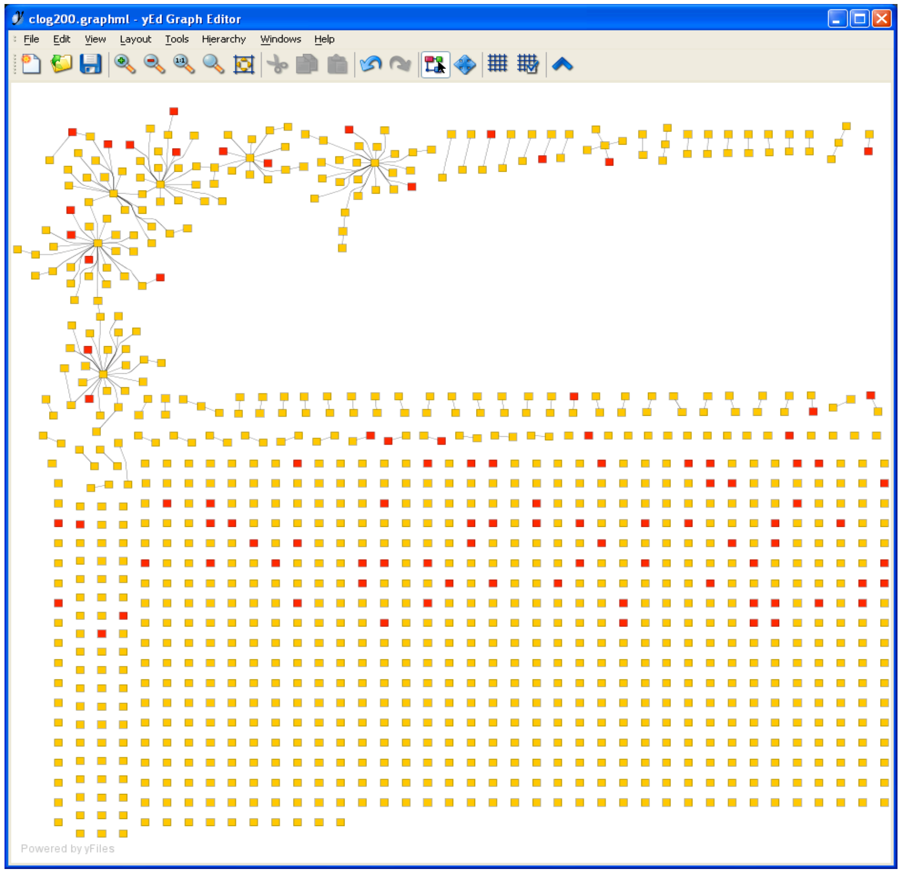
Graph of nodes after 100 connections


clog100.graphml - yEd Graph Editor
File Edit View Layout Tools Hierarchy Windows Help
Powered by yFiles

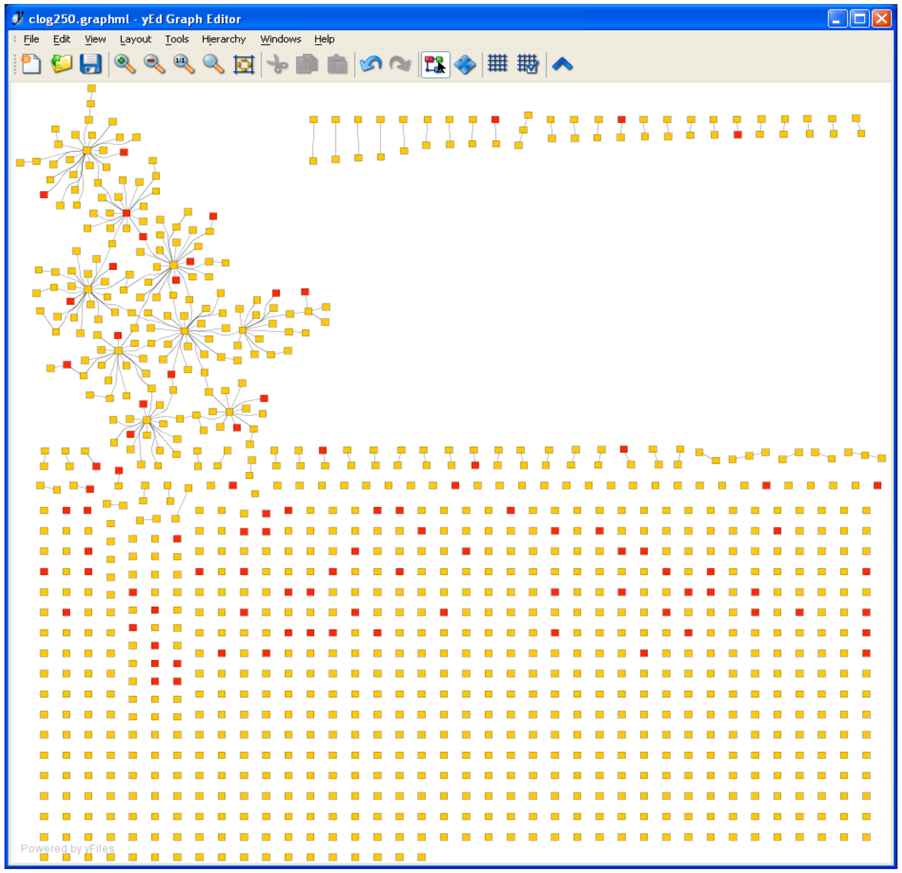
Graph of nodes after 150 connections



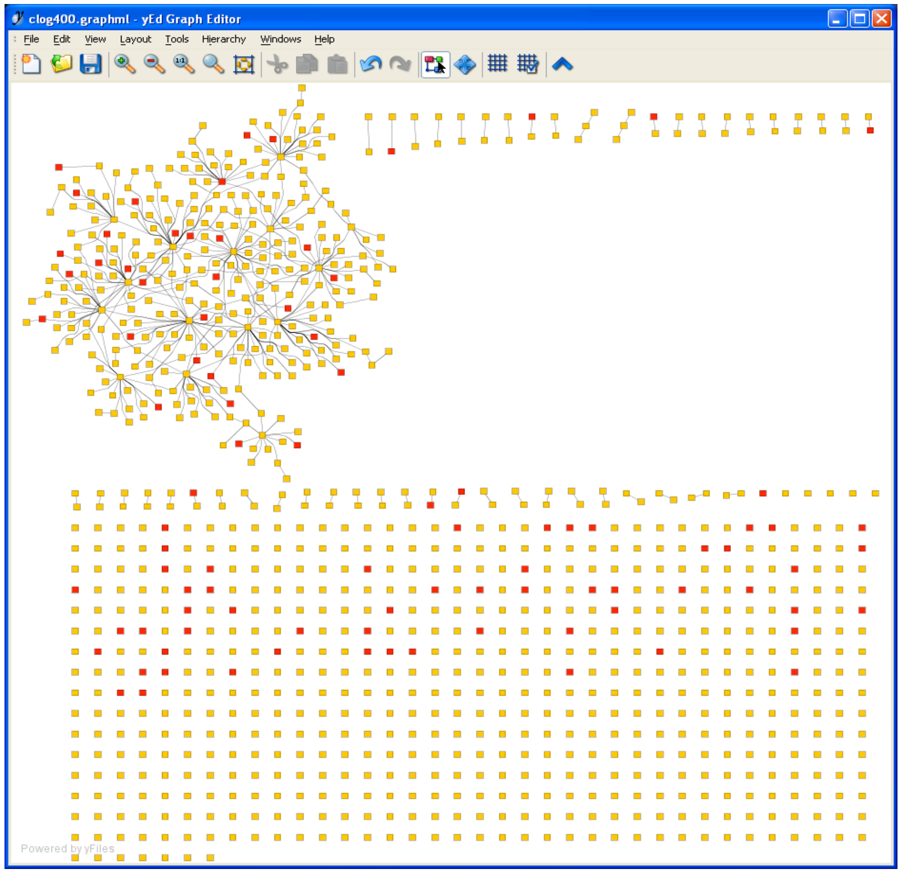
Graph of nodes after 200 connections



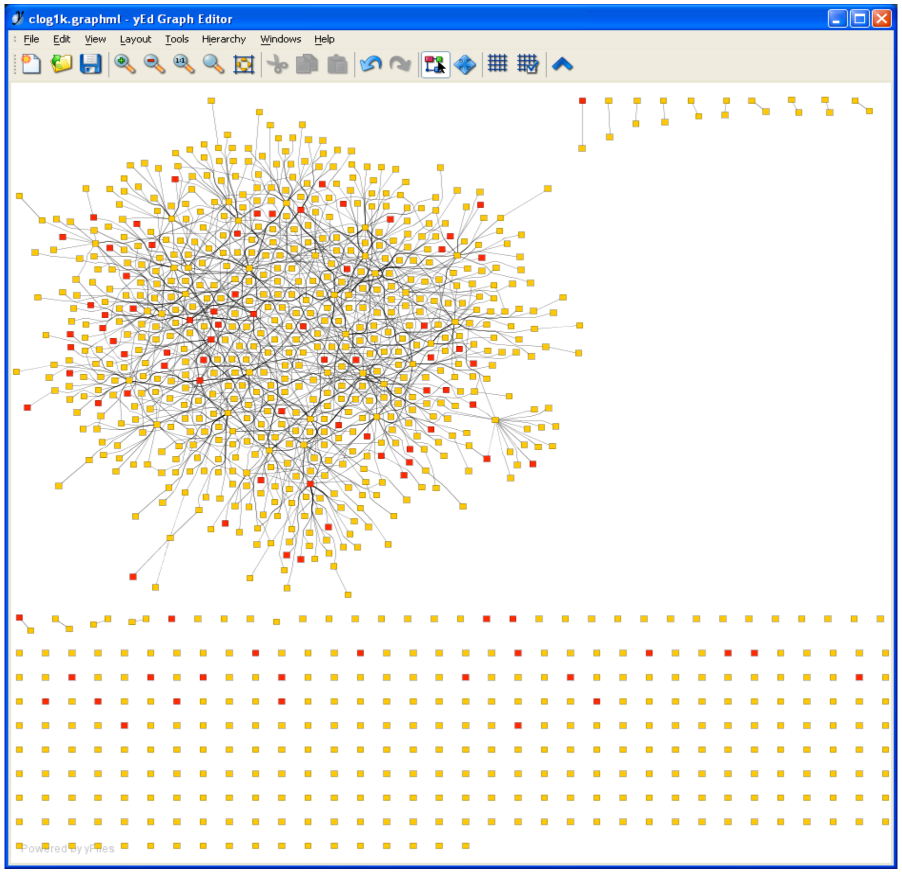
Graph of nodes after 250 connections



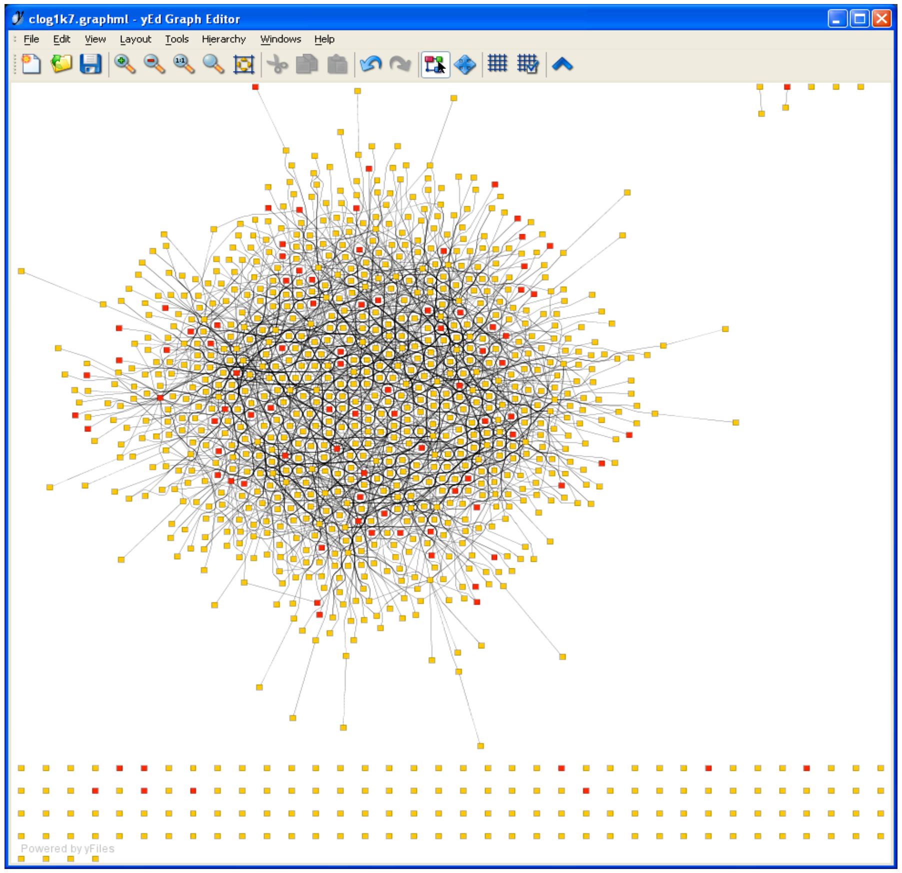
Graph of nodes after 400 connections


clog400.graphml - yEd Graph Editor
File Edit View Layout Tools Hierarchy Windows Help
Powered by yFiles

Graph of nodes after 1000 connections



Graph of nodes after 1700 connections


clog1k7.graphml - yEd Graph Editor
File Edit View Layout Tools Hierarchy Windows Help
Powered by yFiles

Conclusions

- This simulator was used to rapidly develop an executable model of the peer to peer message patterns and central services required to implement a complex Skype based application choreography
- The basic framework described in this paper is concise, efficient, supports large scale simulations, and the results can be visualized.

Works Cited

- [Cygwin] “GNU + Cygnus + Windows”, http://www.cygwin.com/
- [GraphML] “Graph Modeling Language “, http://graphml.graphdrawing.org/
- [Hoare78] C. A. R. Hoare. “Communicating Sequential Processes.” Communications of the ACM 1978.
- [Hoare85] C. A. R. Hoare. Communicating Sequential Processes. Prentice Hall, 1985.
- [KROC]. “Kent Retargetable Occam Compiler” http://www.cs.kent.ac.uk/projects/ofa/kroc/
- [Occam2] Inmos Ltd, Occam 2 Reference Manual. Prentice Hall, 1988.
- [Milner99] Robin Milner, “Communicating and Mobile Systems: the Pi-Calculus”, Cambridge University Press, 1999.
- [Skype]. “Skype Developers API” https://developer.skype.com
- [SSDL] Webber et al. “Asynchronous Messaging between Web Services using SSDL.” IEEE Internet Computing Vol10, No1 2006.
- [WS-BPEL]. “OASIS Web Services Business Process Execution Language” http://www.oasis-open.org
- [WS-CDL] “Web Services Choreography Description Language”, http://www.w3.org
- [yEd] “Graph visualization and editing tool”, http://www.yworks.com/