Improving the Performance of IPOP
Research Project 2

Supervisors:
Ana Oprescu
Kaveh Razavi
Kyuho Jeong
Renato Figueiredo

Dragos Laurentiu Barosan
dragos.barosan@os3.nl
IPOP

• IP over P2P
• Creates links between users leveraging online social connections
• Can bypass NAT
• Secure links
• It supports existing applications
• Libjingle is used for packet forwarding

Motivation

• IPOP allows users to establish connections in cloud infrastructures

• Performance is bad
  • 260 Mbps average throughput with IPOP
  • 950 Mbps average over direct link

• Performance improvement could enable larger adoption
Research Questions

• What are the sources of the performance overhead?

• What are the solutions?
Starting Ideas

• IPOP assumes that connections are always over insecure networks

• IPOP was not developed with performance in mind
  • Possible inefficiencies in the code
Security Performance

• Uses DTLS as security

• Measurements show increase of ~100% when security is disabled
  • 550 Mbps average throughput for an unsecured connection
  • 260 Mbps average throughput for DTLS connection

• Cloud Infrastructure use case requires security for a small number of peers
  • Security cannot be enabled selectively for each peer
  • A more granular approach is better
Enabling Selective Security 1

• Easy solution
  • Each IPOP node has an IPOP interface with an associated IP
  • In the local controller configuration file add the IP’s associated with the peers with which security should be enabled
  • The list of IP’s is checked when creating the link
  • Does not scale
  • It is possible that the IP is not known
Enabling Selective Security 2

- Define a set of groups in the controller configuration file
- Security is enabled if the intersection of the sets is not empty
- Encode group information in “con_req” and “con_resp” messages
Improving Code Performance

Measurements

• Analyze where time is spent by the processor
  • All debugging symbols were enabled
  • Oprofile
    • Kernel and libc symbols
    • Source code annotated with usage percentage
  • Zoom
    • Presents a top down callgraph

• CPU load measurements

• Timing measurements in the code
Receiver bottleneck

• Oprofile
  • ~33 million samples in the receiver
  • ~16 million samples in the sender

• Core on which the receiving thread executes is at ~100% on the receiver side
Receiving Packets in IPOP

- Receiving Thread introduces serialization
- Writing to the tap interface is synchronous
Solutions

• Implement the Producer-Consumer pattern
  • Reading is faster than writing => The writing thread does not wait
  • First implementation with no mutex
    • Use conditional signals as a refinement

• Implement asynchronous writes to save time
  • Linux offers two possibilities
    • POSIX AIO – creates multiple writing threads
    • Libaio – actually queues up write requests in the kernel
Current status & Conclusion

Improve performance up by a factor of two and more to come...

• Users have the possibility of a granular security option
• Analysis shows where time is consumed
• Implementation of more efficient packet processing
Future Work

• Find and fix possible bugs
• Investigate other performance bottlenecks
• Discover new use cases for IPOP