Implementation of a Common API for Transparent Hybrid Multicast - Design & Performance Aspects -

Matthias Wählisch, Thomas C. Schmidt
Stig Venaas

{waehlisch, t.schmidt}@ieee.org,
stig@cisco.com

Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Freie Universität Berlin

link-lab

cisco

Agenda

1. Background: Draft common multicast API
2. Design Aspects
3. Performance Evaluation
4. Open issues
5. Conclusion

Current Multicast Diversity

Group communication services exist in a variety of:

- Different flavors
  - Any Source Multicast vs. Source-specific Multicast
- Different technologies
  - IPv4 vs. IPv6, multicast tunnels, etc.
- Different layers
  - Native multicast vs. overlay distribution

Implications

- Programmers decide on technology at coding time
  - How do they know about the multicast deployment state at this time?
- Applications provide their own solutions to overcome inter-domain deployment problem
  - Increases complexity & introduce redundancy
- Lack of efficiency
  - Reasonable to assume no global IP-layer multicast

➢ Difficult to write multicast applications that run everywhere & use most efficient group service

Recall: Common Multicast API Draft

Idea: Move complexity from application to the system level

The current draft provides ...

- a common multicast API on app. layer that abstracts group communication from distribution technologies
- abstract naming and addressing by multicast URIs
- mapping between naming and addressing
- definition of protocol interaction to bridge multicast data between overlay and underlay

Status

- Version 00/01 presented at IETF 76, Hiroshima
- Update version 02 presented at IETF 77, Anaheim
  - Interesting work, but needs extended motivation
- Current version: 04
  - Version 03 submitted before 78th IETF
- Version 05: Working on an update, which also includes insights of the NBS BoF
  - Will be available some weeks after this IETF meeting

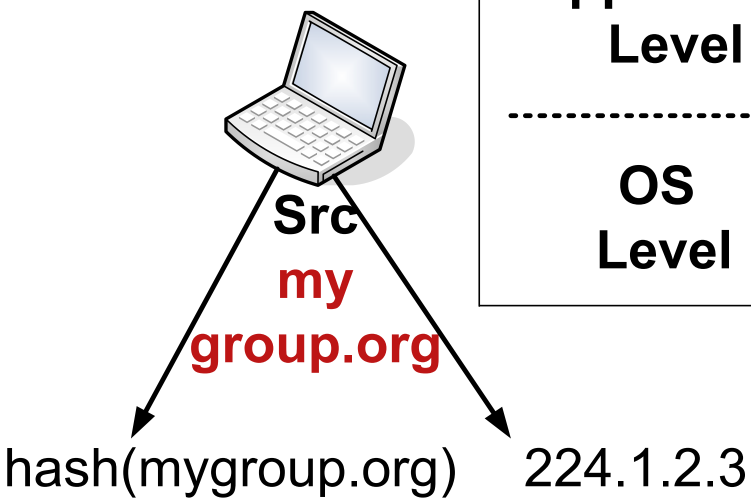
Common Multicast API & Middleware

Application Level

```
MulticastSocket msock =
        new MulticastSocket();
msock.send(data,"mygroup.org");
```

OS Level

1. Mapping of Group Name to Addresses
2. Send data to technology specific interfaces



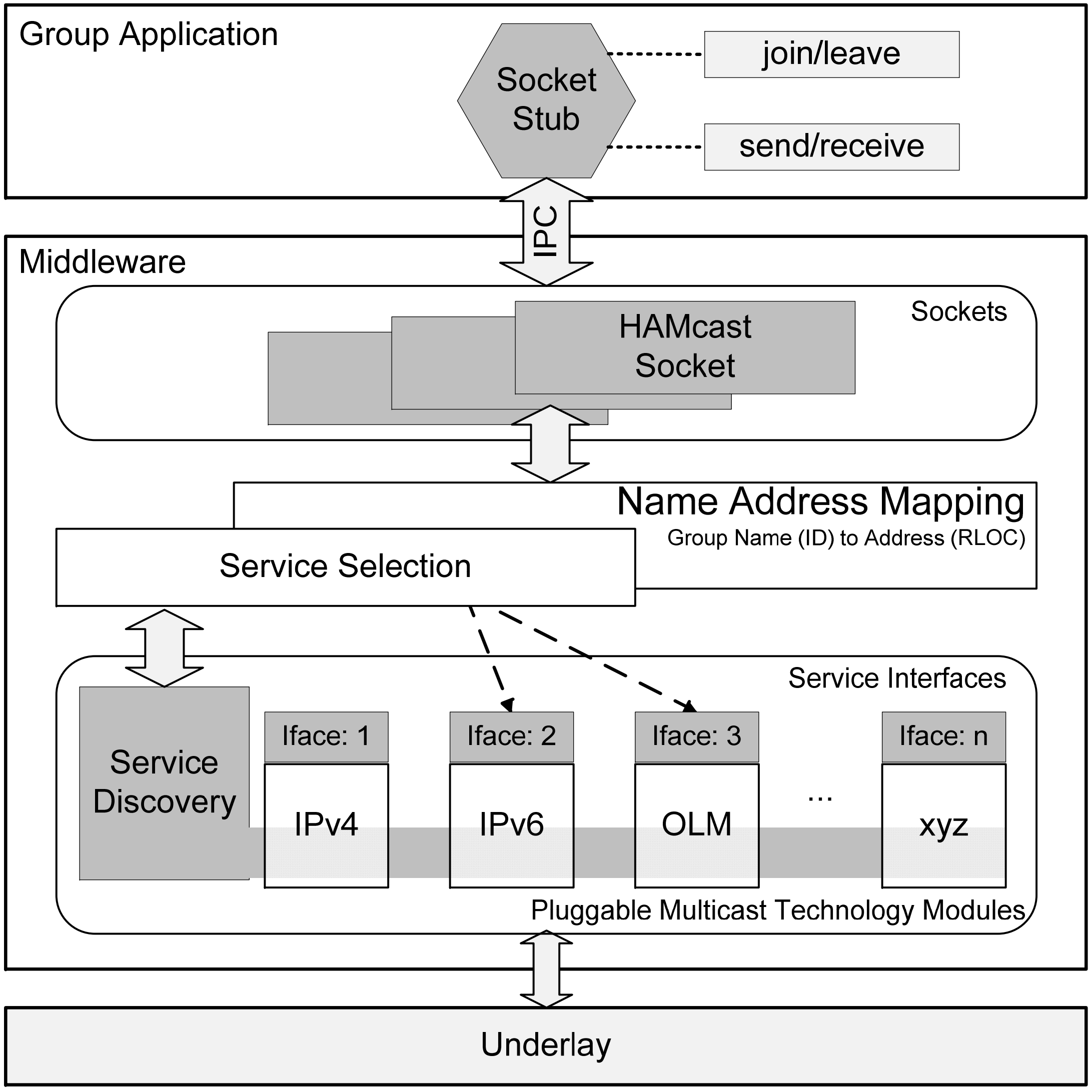
- Mapping of names to technologies at run-time
- Late binding

Implementation Report: The HAMcast stack

Design Aspects:

- Flexibility for future extensions and adaption to new network technologies
  - No recompilation or changes to existing application using the HAMcast stack
- Easy integration of new programming languages
  - Only mapping of API calls & 'signaling' to OS layer
- Separate implementation of general multicast logic from technology-specific multicast instantiation

Overview HAMcast Stack



Middleware: Rough Overview

- Unique daemon process instantiated once per host
- Implemented in C++
- Service functions realized by single modules
  - Can be loaded by the system if required
- Technology-specific service interfaces
  - Implement a specific multicast understanding, e.g., IPv4/6, OLM, ...
  - Includes service discovery
- Send/receive functions implemented as asynchronous calls

Module Example: Service Discovery

- Implemented by each technology module
- IP-layer multicast:
  - Discovery based on libcap sniffer library
  - Observation of general IGMP/MLD queries & PIM Hello
- DHT-based multicast:
  - Unicast overlay is part of HAMcast stack
  - Inquires group and neighbor states
- Gateways (e.g., AMT):
  - Future work

Prototype Status

- First working version presented at EuroView'10
- Hybrid distribution of streaming video using IPv4 + Scribe
- Monitoring tool to discover and visualize hybrid multicast tree structure
  - Usage of API service calls

Performance Evaluation

Aim:

- Basic evaluation of stack implementation
  - We do not focus on distribution technology
- Comparison with native unicast

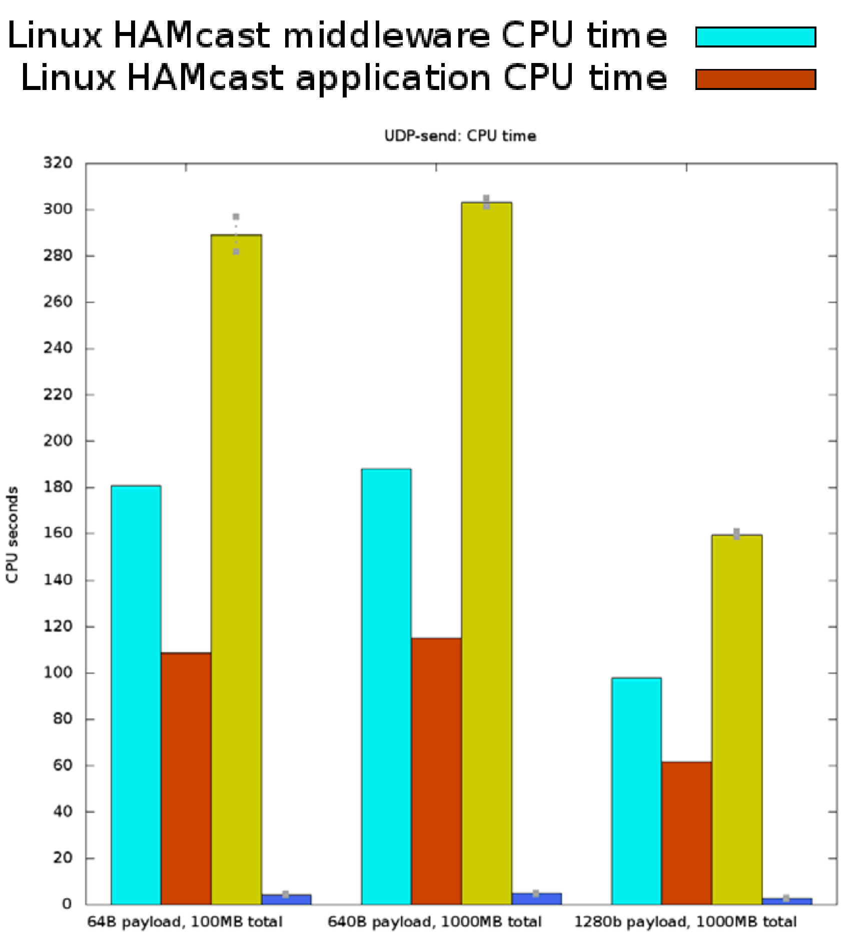
Setup:

- Single source, single receiver scenario; Linux OS
- Constant bit stream; 100 Mb/s Uplink
- Different packet sizes
- Average over several runs until convergence

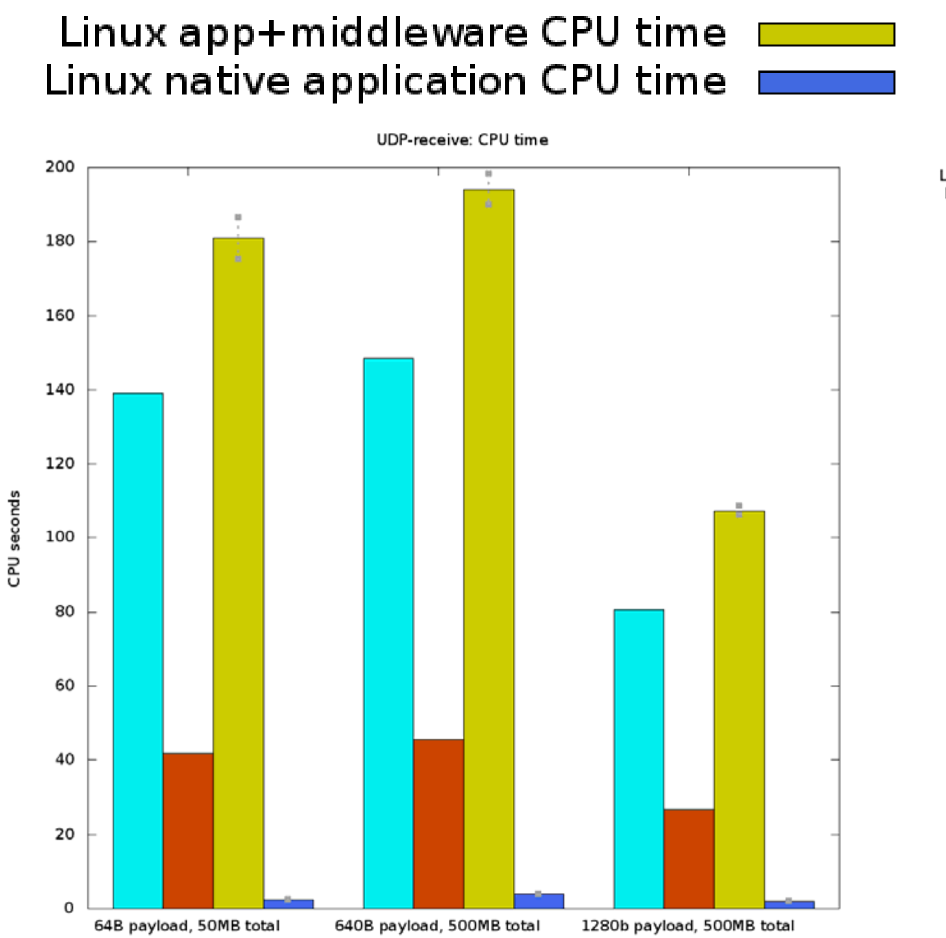
Caveat

- **Ongoing** evaluation & optimization
- Performance results are a **first** view: Optimization still to come

CPU Performance Send/Receive

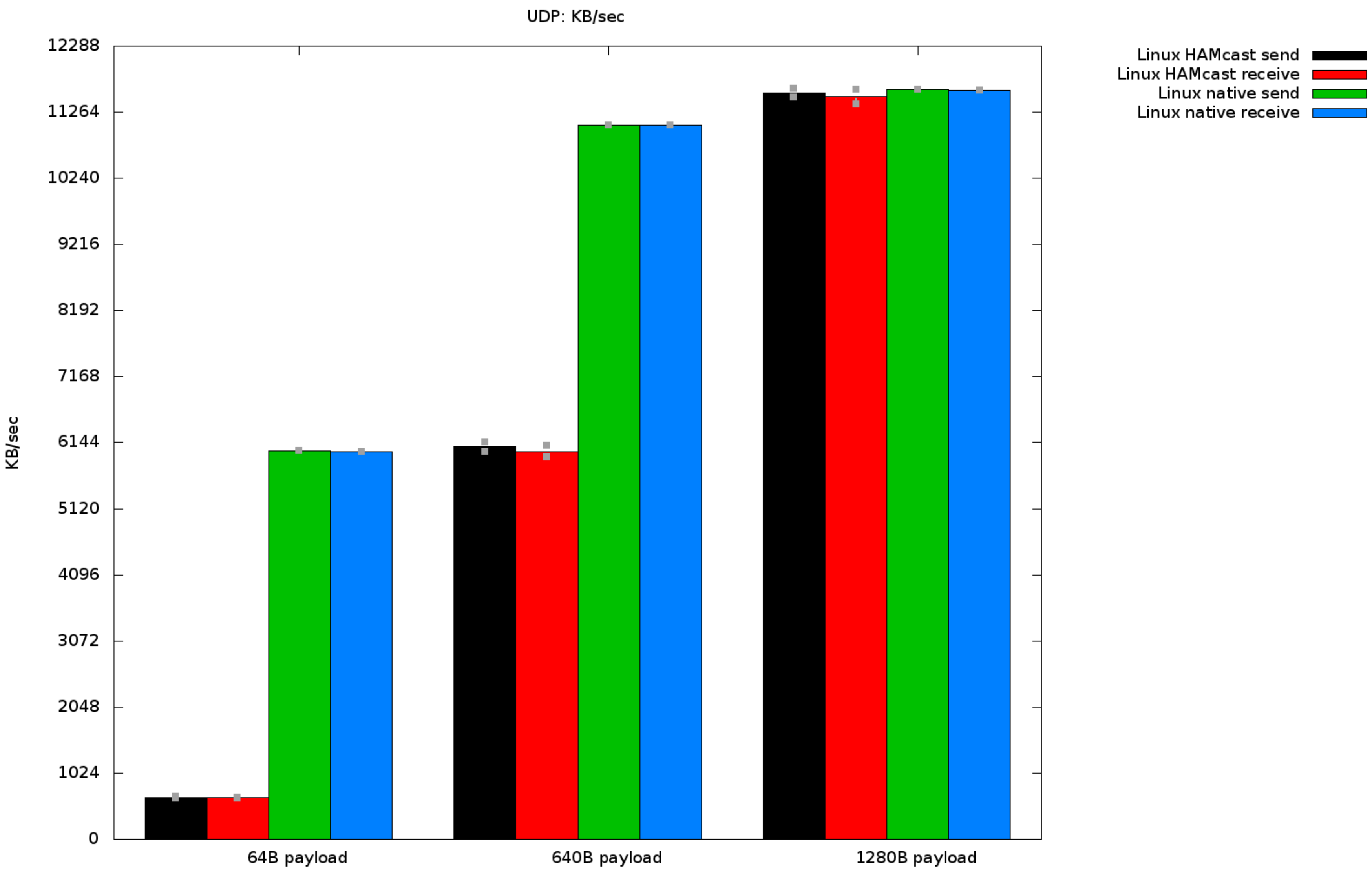


Send

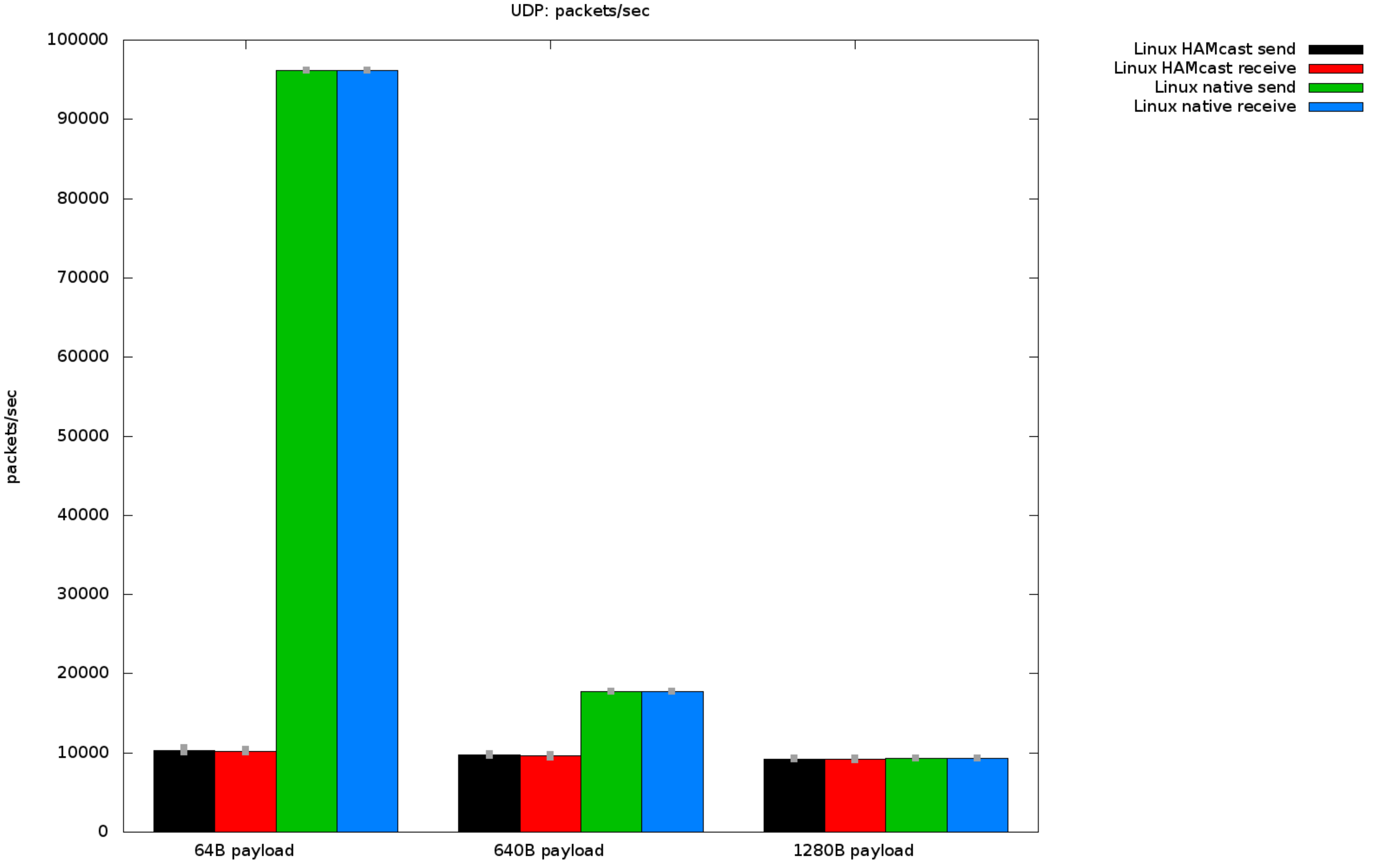


Receive

Data Rate


UDP: KB/sec
12288
11264
10240
9216
8192
7168
6144
5120
4096
3072
2048
1024
0
KB/sec
64B payload
640B payload
1280B payload
Linux HAMcast send
Linux HAMcast receive
Linux native send
Linux native receive

Packet Rate



UDP: packets/sec
packets/sec
100000
90000
80000
70000
60000
50000
40000
30000
20000
10000
0
64B payload
640B payload
1280B payload
Linux HAMcast send
Linux HAMcast receive
Linux native send
Linux native receive

HAMcast stack at a Glance

- Middleware implemented in C++
- Programming libraries for C++ (and Java)
- IPv4 and IPv6 module for native IP-layer multicast
- Scribe implementation for overlay multicast
  - Based on Chimera P2P network stack
- Service discovery for IPv4/v6 multicast & OLM
- Prior to optimization: Moderate performance results

Open Issues

- More detailed analysis of stack performance
- Optimize implementation of HAMcast stack
- Development of libraries for further programming languages
- Service selection strategy:
  - Prefer most efficient technology
  - Do not distribute in parallel
- SSM support

Thank you ...

- Please, read the upcoming version 05
- Interest in the implementation project?
- More feedback is needed by RG members!