RI2N/UDP: High bandwidth and fault-tolerant network for a PC-cluster based on multi-link Ethernet

Takayuki Okamoto, Shinichi Miura, Taisuke Boku, Mitsuhisa Sato, Daisuke Takahashi

Graduate School of Systems and Information Engineering, University of Tsukuba
Outline

- Motivation and background
- Design and implementation
  - System construction
  - Protocol
- Performance evaluation
  - Throughput, fault tolerance, latency
- Related works
- Conclusion and future work
Interconnection used in clusters

- SAN (System Area Network)
  - High bandwidth, low latency, high availability, large scalability
  - Expensive for small-scale clusters
- Ethernet
  - High performance/cost ratio
  - Lower performance, availability and scalability

Providing high performance and high availability with multi-link Ethernet only by software

RI2N

**Redundant Interconnection with Inexpensive Network**
Concept of RI2N

- High bandwidth and fault-tolerant network based on multi-link Ethernet
  - High throughput between nodes
    - Data striping
  - Fault tolerant function for clusters
    - Redundant links and switches
Previous implementation of RI2N/TCP

- **User level prototype with TCP/IP**
  - Establishes a TCP connection on each Ethernet link
  - Achieves 230MB/s using dual links of GbE
  - Fault tolerant function was not implemented
  - Problem of TCP behavior when hardware fails
    - Old linux TCP can be blocked for a long time in select() system-call
  - Both TCP and RI2N should care about retransmission

New implementation of RI2N with UDP/IP
Design of RI2N/UDP

- **User level implementation**
  - High portability for various operating systems and hardware

- **Implementation with UDP/IP**
  - Using multiple subnet with private IP address space
  - Simple as like as IP, a protocol which doesn’t care any packet-losses under it
  - Suitable to implement new protocols on it

- **High bandwidth and fault tolerance**
  - Data striping
  - Use of redundant links

- **TCP compatible socket API**
  - Easy to port existing applications for TCP
  - Guarantee the reliability
    - Acknowledgement, serialization, retransmission
    - Dedicated RI2N thread for asynchronous processing
System construction (1)

- **TCP compatible API library**
  - Called from application

- **RI2N communication thread**
  - Created by initialization function in API library

- **User thread**
  - User application keeps running as user thread
- **Two data streams**
  - Data sharing with user thread
  - UDP stream on multiple NICs

- **Efficient handling and high throughput**
  - `select()` for thread synchronization with socketpair
  - Shared buffer for data transfer with high throughput
RI2N/UDP protocol

- **Packet transmission to multilink**
  - Using data striping with round-robin manner
  - Assuming: a homogeneous multi-link

- **Retransmission control**
  - Multi-ACK packing to reduce the number of packets
  - Selective acknowledgement by bitmap

- **Flow control**
  - Simple window control

- **Fault tolerance**
  - Failure and recovery detection
Retransmission control (1)

- **Multi-ACK packing to reduce the number of ACK**
  - Same technique as TCP acknowledgement
  - Each packet has an *ACK-id* pointing the packet which should be received next
  - Frequent packet racing causes a lot of retransmission

```
<table>
<thead>
<tr>
<th>Non-flagged</th>
<th>flagged</th>
</tr>
</thead>
<tbody>
<tr>
<td>1002</td>
<td>1006</td>
</tr>
<tr>
<td>1003</td>
<td></td>
</tr>
<tr>
<td>1004</td>
<td></td>
</tr>
<tr>
<td>1005</td>
<td></td>
</tr>
</tbody>
</table>

“Flagged”: it requests for remote side to send back ACK packet

<table>
<thead>
<tr>
<th>( non-flagged )</th>
<th>Nothing to do</th>
</tr>
</thead>
<tbody>
<tr>
<td>( flagged )</td>
<td>ACK is returned soon</td>
</tr>
</tbody>
</table>

“OK before #1006”

sender

receiver
Retransmission control (2)

- Large buffer for serialization
- Selective acknowledgement

- Bitmap represents the packets which have been already received and whose seq number are larger than ack id

![Diagram of retransmission control with a receive buffer and an acknowledgment bitmap.](image-url)
Flow control

- **Simple window base flow control**
  - Less packet losses on cluster network
  - To utilize the max throughput of hardware
  - Receiver publishes the window-size
  - Window size = buffer size remained on receive side

  Burst transfer based on window size

  The last packet in a window is flagged to request ACK

  Restart transmission

  Update window size

  ACK

  Data of 3 packets is read form user thread

  Free area of buffer

  Receiver
Policy of failure and recovery detection

- **Failure detection**
  - Significant degradation of throughput
    - Throughput falls to 1 to 10% of max throughput
  - Failure should be detected *as soon as possible*

- **Recovery detection**
  - Throughput keeps certain throughput after recovery
  - Recovery requires reconnection or replacement which takes few minute at least
  - Recovery does not have to be detected within seconds
Failure detection

- **Comparing the number of received packets**
  - If the count of packets is largely differ from the others, the link can be considered as failure
  - Sharing the information between a pair of RI2N sockets
  - Omitting the failed link from the target list

Omitting the link

Considering as failure
Recovery detection

- Sending heartbeat packets on all links at an interval of few seconds
  - If a heartbeat is received at the link marked as “failed”, the link can be considered to be recovered
  - Adding the link to the target list
Performance evaluation

- Intel-Xeon 3.0GHz dual-channel DDR2-400 1GB
- Linux 2.6.17
- NIC driver e1000 7.0.33
- Dell PowerConnect 5224 (24ports GbE-switch)
- Intel PRO/1000 MT dual port

DAC2007
Throughput before and after failure

**Graph Analysis:**
- Link 1 failed at time $t_1$.
- Link 2 failed at time $t_2$.
- Link 1 was recovered at time $t_3$.
- Link 2 was recovered at time $t_4$.

**Observations:**
- Keeping certain throughput despite link failures.
- Resuming transmission after link recovery.

**Conclusion:**
RI2N/UDP can keep communication on a partial link failure.
Maximum throughput

Dual linked RI2N/UDP provides two times of throughput of TCP

- Maximum throughput: 243MB/s
- TCP throughput: 123MB/s
Latency

RI2N/UDP is suitable to an application using large messages.
An analysis of latency

- Detailed time required for RI 2N/UDP processing
- Latency (a half of RTT) was 72µs
  - Inter-node: 30µs (red) = 30
  - Intra-node: 40µs (blue) = 35+1+4
  - 30+40 = 70 ≈ 72µs

Comm. between threads dominates the latency

Changing the inter-thread comm. method to use only socketpair → −5µs

Considering “non-thread” implementation ⇒ “Driver”
Related works

- **Link Aggregation (IEEE802.3ad) – driver level**
  - Provides high bandwidth and fault tolerance based on multilink Ethernet
  - Single switch pair → non-scalable, non-supporting for switch failure
  - RI2N/UDP supports multi-paths & multi-switches

- **VMI2.0 – socket level**
  - A socket layer supporting multilink network
  - Supports some heterogeneous interfaces for fault tolerance
  - RI2N/UDP also provides data striping to increase the bandwidth

- **Open MPI, LA-MPI – library level**
  - An MPI implementation supporting multilink network
  - Fault tolerance will be available in Open MPI v1.2
  - RI2N/UDP provides a socket level interface → wide applications
Conclusion

- **Implementing RI2N on UDP/IP**
  - An user level library for high bandwidth and fault tolerance
  - Just using commodity network (Ethernet) and software

- **Performance Evaluation**
  - Higher bandwidth
    - Dual linked RI2N/UDP achieved exactly two times of throughput for GbE
  - Fault tolerance
    - Keeping the communication while a part of links failed
    - Automatically detecting the recovery and restarting

**RI2N/UDP provides high bandwidth and fault tolerance for PC cluster only with commodity network and software**
Future works

- To improve the flow control algorithm
  - Current algorithm is too simple to be used for multi-point comm. on a network which has a bottleneck on the path

- RI2N/UDP under MPI
  - We will use MPICH/ch_p4
  - Practical evaluation with some MPI applications is required
    - Especially with an application hiding the communication time by computing time

- To restrict the comm. method between thread only with socketpair
  - Reduces the latency between threads
  - Provides higher compatibility to TCP socket
    - It is easy to implement some socket functions (etc. select(), poll())

- Another implementation in pseudo device driver level
  - Provides the latency as same as TCP/IP