TCP Connection Passing

http://tcpcp.sourceforge.net/

Werner Almesberger
werner@almesberger.net

July 17, 2004

Overview

- Introduction
  - History and goals
  - Possible uses
  - Alternatives
- Technical details
  - The connection passing procedure
  - Network traffic switching
- Advanced issues and future work
  - Testing
  - Congestion control
  - Checkpointing

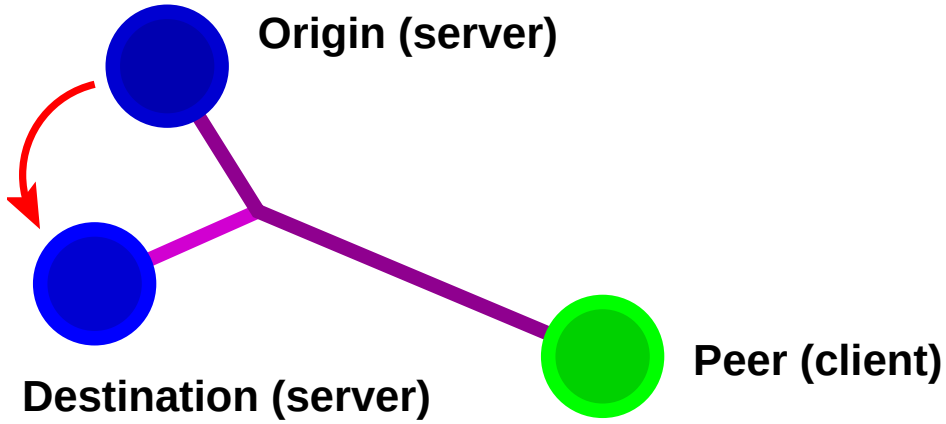
How it all began ...

Linux-Kongress 2002, a talk on fault tolerance by Fábio Olivé Leite (from Conectiva, Brazil):

- Moving an IP address from box A to box B is easy, but
- TCP connections need to be restarted
- **Why ?**

Goals

- Pass "live" TCP connection from one host to another
- Transparent to peer
  $\rightarrow$ no wrapper library, proxy, etc.
- As transparent as possible to application (on host)



Possible uses

- Process migration
  E.g. openMosix
- Connection dispatcher
  E.g. send to least loaded Web server
  or authenticate, start encryption, pass to "dumb" server
- Checkpoint/restart
  Limitation: must stop and restart immediately
  E.g. software update
- Failover
  Limitation: needs prior warning
  E.g. fan failure

Alternatives (1)

- Break the connection, let peer fix it
  - ☺ Works with existing infrastructure
  - ☹ Delays, or even lack of automatic recovery (SSH, X11)
  - ☺ Users may not expect any better
  - ☹ Hard to augment state information
- Redirection at higher-level protocol (e.g. HTTP)
  - ☺ Works with existing infrastructure
  - ☹ Not every protocol supports this
  - ☹ Not possible in every state (HTTP)

Alternatives (2)

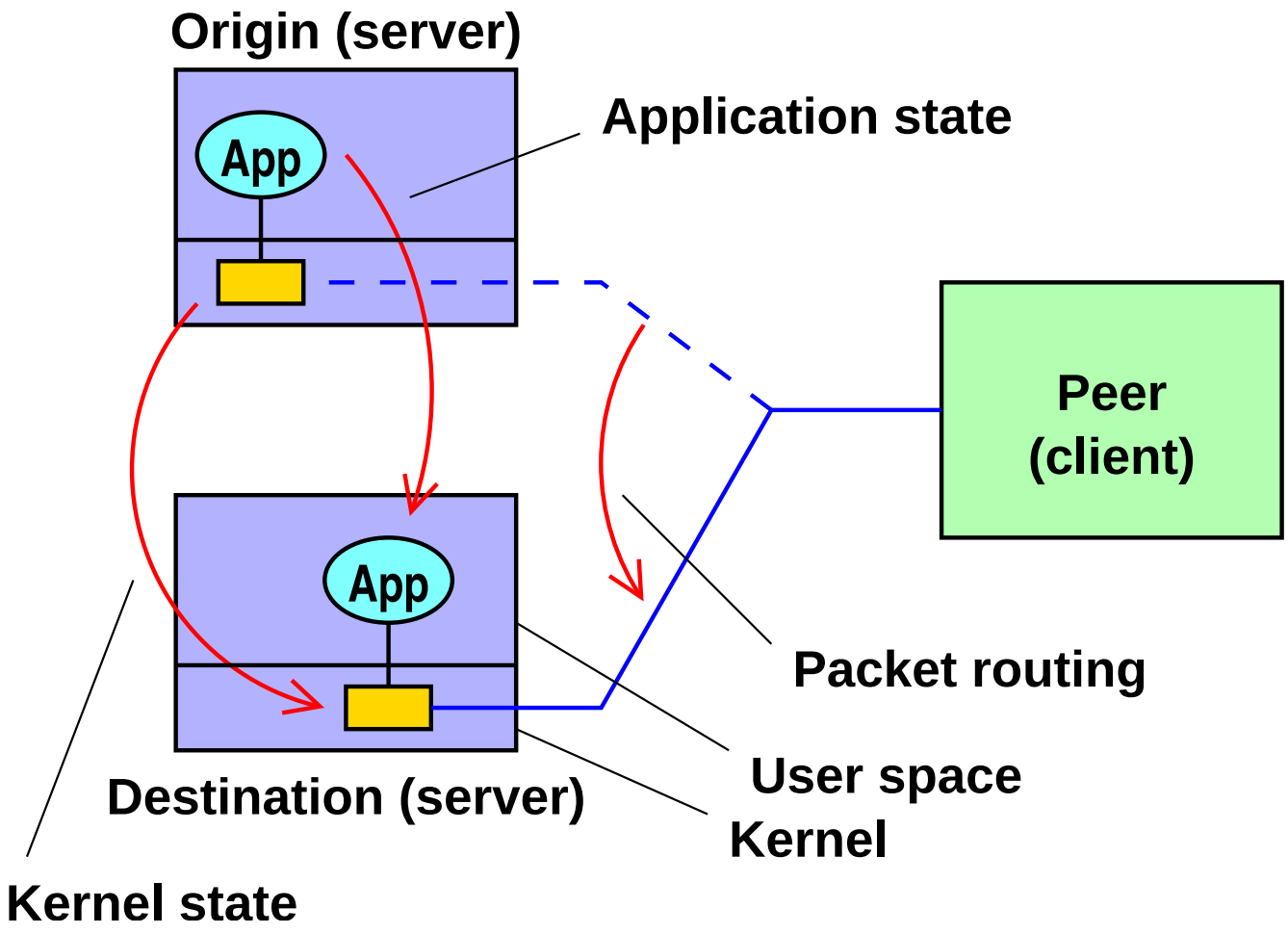
- Intermediate layer with own transport
  - ☺ Maximum flexibility
  - ☹ Peer needs to talk the same protocol
  - ☹ Integration may be tricky (API changes)
- Use a TCP (or above) proxy/agent
  - ☺ Hides *any* mechanism from peer
  - ☹ Proxy more complex than basic router or NAT box
  - ☹ Proxy may become bottleneck

Technical details

- Application state
  - Static file server: File position
  - Search engine: Query string
  - Process migration: Entire process context
- TCP connection state (kernel)
- Path through network
  - Routing, NAT, ...

tcpcp only addresses kernel part
→ building block for complete solution

Switching: high-level overview



TCP items to move (1)

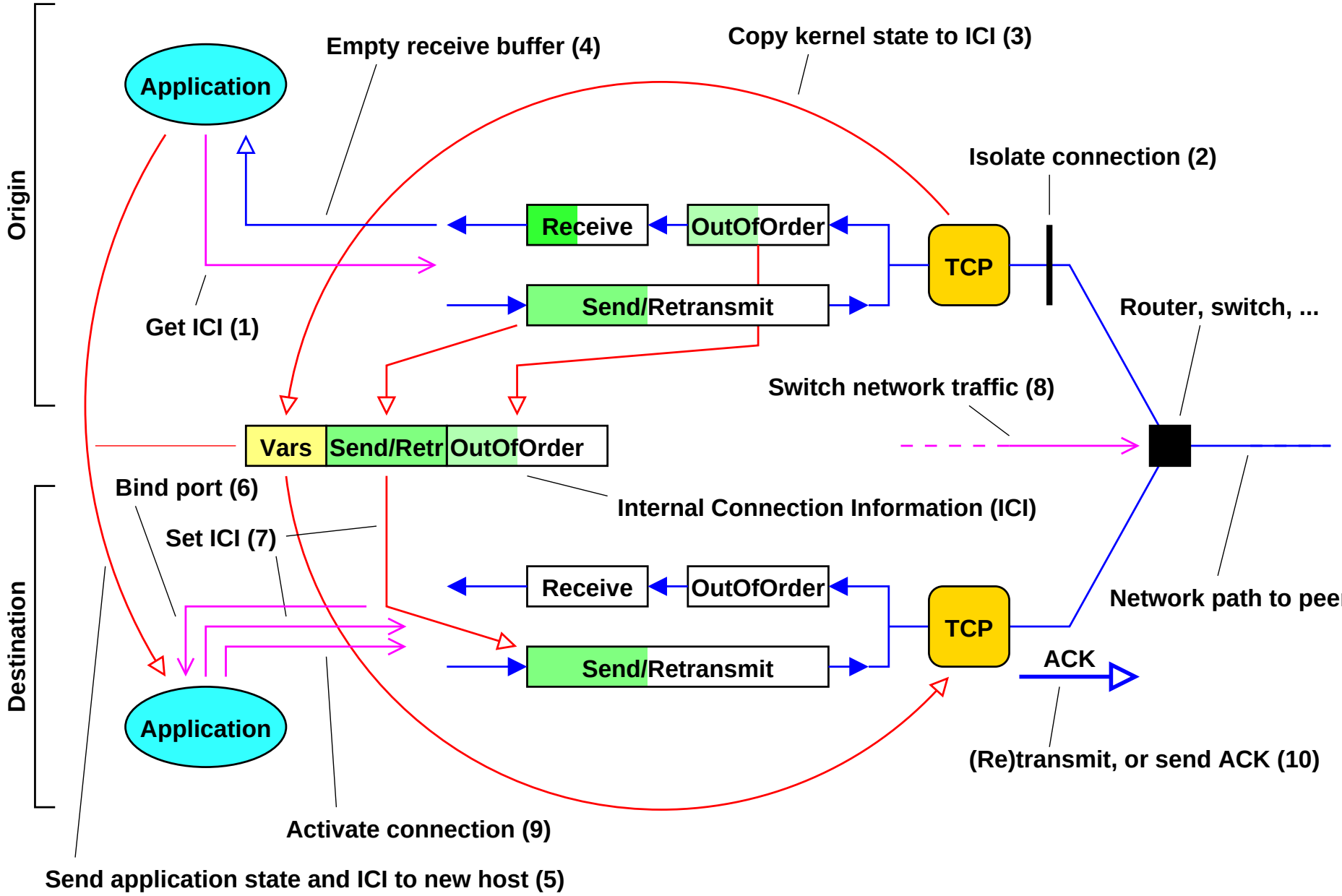
- Connection identifier
  - Source/destination IP address
  - Source/destination port number
  - If using NAT: allow changing the local end
- Negotiated capabilities
  - Window scaling
  - MSS (Maximum Segment Size)
  - Timestamps
  - SACK (Selective ACKnowledgments)

TCP items to move (2)

- Sequence numbers (main synchronization)
- Window
  - Flow control (how much more can I send ?)
  - Window cannot shrink on its own
- Timestamps
  - Tricky: kernel uses central timestamp source
    $\rightarrow$ need to introduce offset from local time
- Send/retransmit buffer
  - Data is no longer visible to application

Items we can ignore (for now)

- Receive buffer
  - Application must empty buffer while switching
  - Can choose to process or to pass data
- Out-of-order buffer (incomplete data)
- Loss recovery state
  - Lose all data not guaranteed
  - SACK allows us to change our mind (outbound)
  - SACK can be forgotten (inbound)
- Congestion control state
  - Return to slow-start
  - Conservative, but possibly inefficient


Origin
Application
Empty receive buffer (4)
Copy kernel state to ICI (3)
Isolate connection (2)
Receive
OutOfOrder
TCP
Send/Retransmit
Get ICI (1)
Router, switch, ...
Switch network traffic (8)
Vars
Send/Retr
OutOfOrder
Bind port (6)
Internal Connection Information (ICI)
Set ICI (7)
Destination
Receive
OutOfOrder
TCP
Network path to peer
Send/Retransmit
ACK
Application
(Re)transmit, or send ACK (10)
Activate connection (9)
Send application state and ICI to new host (5)

Switching network traffic (1)

Requirements:

- Must work at the IP level, or below
- Must be quick
- Should scale for bandwidth
- Should scale for redundancy (parallel paths)
- Should be easy to implement and maintain

Switching network traffic (2)

- Use local routing
  - ☺ Fast and easy to do (uses existing equipment)
  - ☺ Can be distributed
  - ☹ Poor granularity (moves all connections at once)
  - ☹ Creates single *area* of failure

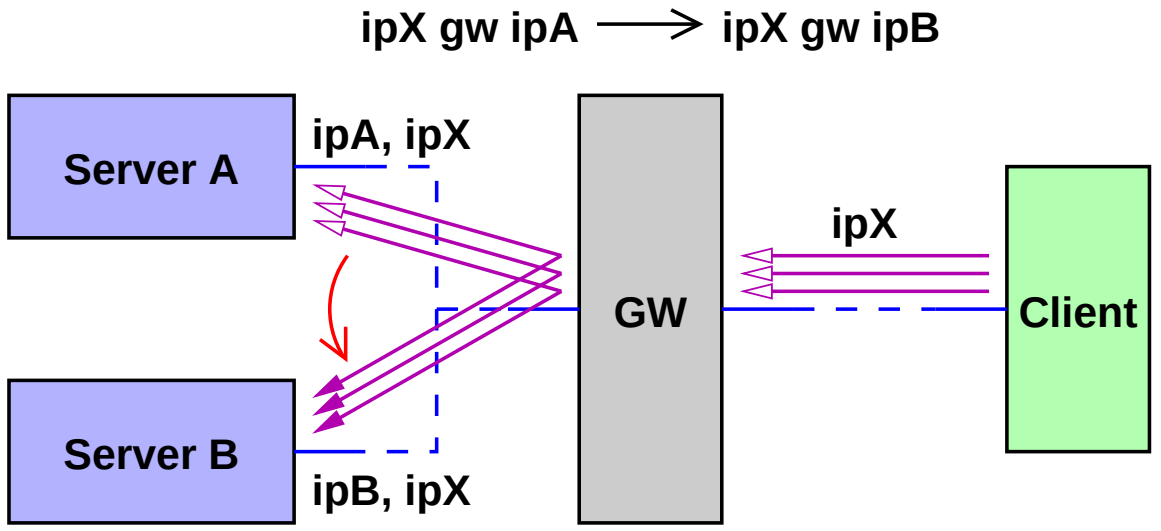
Initial setup:

`route add` *ipX* `gw` *ipA*

Switch:

`route del` *ipX*

`route add` *ipX* `gw` *ipB*



Switching network traffic (3)

- Use a shared media
  Similar to local routing, but doesn't need access to the router(s)
- Use global routing ?
  - ☺ Distributed and scalable
  - ☹ Hard to do, maybe even "outlawed"
  - ☹ Poor granularity (moves all connections at once)
  - ☹ Transition is slow to begin
  - ☹ Transition takes a long time to complete (5 . . . 60 minutes)

Switching network traffic (4)

- Use a NAT
  - ☺ Can move individual connections
  - ☺ Each connection can have its own destination
  - ☹ Consumes more resources than routing
  - ☹ Creates single point of failure (maybe)
  - ☹ State synchronization with host is tricky
    (static NAT vs. connection tracking)
- Tunnels, VPN, Mobile IP, Anycast, ...
- Invent your own
  - ☺ The world is full of creative abuses of TCP/IP
  - ☹ Not all of them are nice

Testing (to do)

Testing is fun, because ...

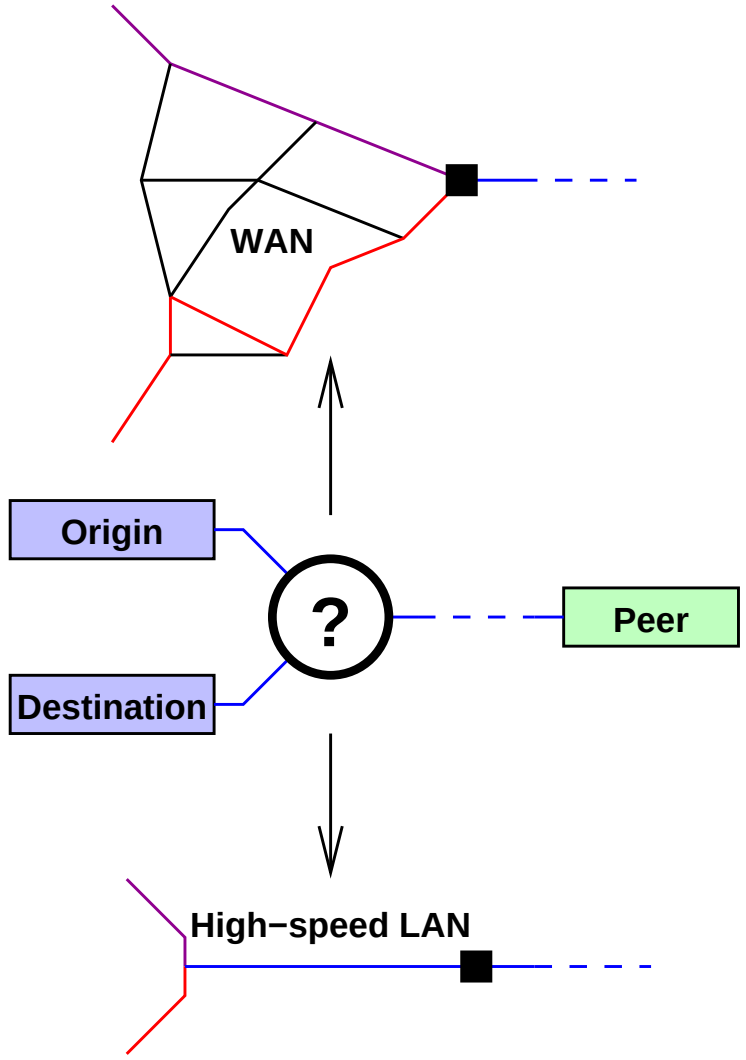
- Lots of transient states (error recovery, etc.)
  Almost impossible to reproduce
- TCP/IP stack is complex and changes often
  $\rightarrow$ needs frequent re-testing

Solution:

- Build regression test suite
- Use umlsim
  `http://umlsim.sourceforge.net/`

Congestion control

- "Guess" bandwidth along path
- Fall back to slow start
  - Conservative
  - Path is unknown
  - Path known to differ
- Keep on going
  - Aggressive (dangerous)
  - Path known to be equal



Checkpointing and failover

- Outbound should be feasible
  - We can retransmit acknowledged data
  - Save frequently to avoid sequence number problems
  - Timestamp can be "guessed" if based on real time
  - May retransmit a lot of data
- Inbound may be too intrusive
  - Problem: kernel acknowledges unsaved data
  - Application would have to control when acknowledgments are sent
  - May confuse timing (and congestion control)

Conclusion

- What is done
  - Proof of concept implementation (on 2.6.4)
  - Basic kernel-level functionality reasonably stable
    i.e. building upon it not too risky
- To do (still a lot)
  - Implement NAT-based network traffic switching
  - Begin testing
  - Integration with applications or middleware
- For further study
  - More flexible congestion control
  - Checkpointing for *real* failover

`http://tcpcp.sourceforge.net/`

API: Origin

```
void *buf;
int ici_size;
size_t size = sizeof(int);

getsockopt(s,SOL_TCP,TCP_MAXICISIZE,&ici_size,&size);
buf = malloc(ici_size);
size = ici_size;
getsockopt(s,SOL_TCP,TCP_ICI,buf,&size);
```

..., empty receive buffer, send ICI and application state.

API: Destination

Receive hand-over, extract ICI, then ...

```
setsockopt(s,SOL_TCP,TCP_ICI,
    buf,size);
```

..., switch network traffic, ...

```
int sub_function = TCPCP_ACTIVATE;

setsockopt(s,SOL_TCP,TCP_CP_FN,
    &sub_function,
    sizeof(sub_function));
```

Answers expected from testing

- Does **tcpcp** break the TCP/IP stack ?
- Do changes in the TCP/IP stack break **tcpcp** ?
- Does **tcpcp** work at all ?
  - Connection eventually resumes
  - Doomed connections die
  - Doesn't Oops the kernel, no matter what
- Does **tcpcp** work as planned ?
  - Fast recovery (or failure)
- Do we violate any rules ?
  - Don't send invalid packets
  - Conform to congestion avoidance

Security

- **tcpcp** can be abused to commit some protocol violations
- Creating a connection can be restricted
- Worse things can come from the Internet ...
- See paper for details