Transport Layer Support for Highly-Available Network Services

Florin Sultan, Kiran Srinivasan, Liviu Iftode
DisCo Lab
Rutgers University

Motivation

- Internet: Services instead of Servers
  - Client oblivious to a given server's location
  - Client interested only in the service and QoS
- Internet Services: highly-available and high performance

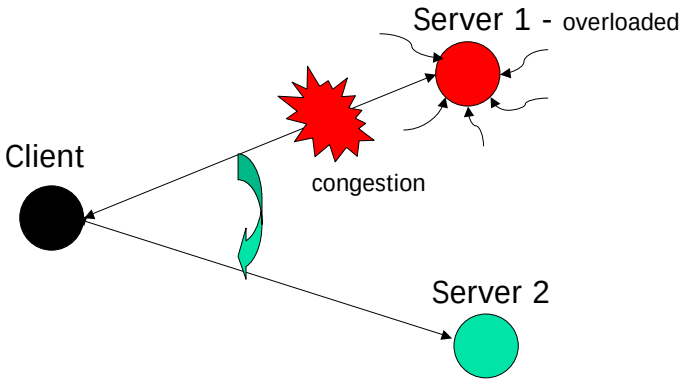
TCP-based Internet Services

- Adverse conditions
  - core network congestion, server overloaded, failed or under DoS attack
- TCP response: keep sending to the same server
  - TCP does not enable clients to dynamically change their connection for better service
- TCP: implicit binding of service to a server

Our Solution: Migratory TCP

- A client connection can transparently migrate to different servers during its lifetime
  - Modified TCP to enable connection handoffs
  - Servers cooperate to support the handoff
  - Client applications do not change
- Servers can be geographically distributed
- General and flexible (not application-specific)

The Connection Migration Model



Related Work

- FT-TCP [Alvisi '00]: failure masking
  - persistent connections across server crashes
- Application-specific solutions
  - LARD [Pai '98]: TCP connection hand-off by a front-end in cluster-based Web server
  - [Yang '99]: HTTP back-end server fail-over
    - front-end stores state for all connections
  - [Snoeren '00]: HTTP server fail-over by connection migration
    - soft TCP and HTTP state maintained at back-up servers

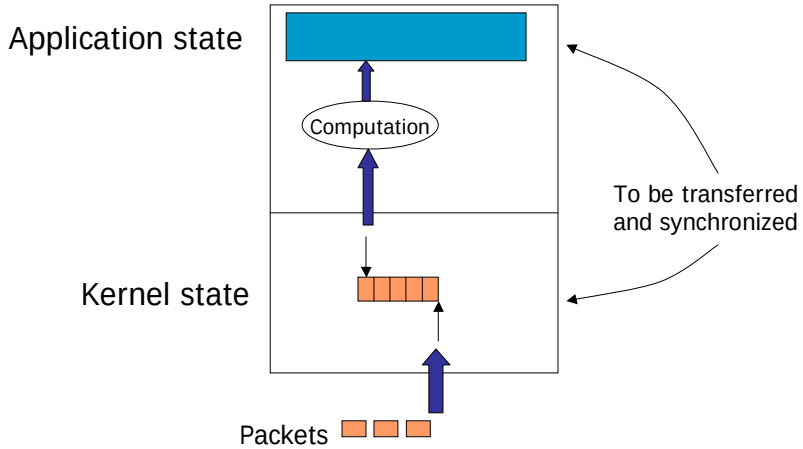
Related Work (cont'd)

- Stream Control Transmission Protocol (SCTP)
  - supports data exchange between *two* endpoints
  - multi-homing (many IP addresses / endpoint) ensures path redundancy
    - failures on local network - use redundant LANs for access
    - failures in core network - exploit alternate routing paths to remote endpoint

Implementation

- Per-connection application state (fine-grained migration)
  - we do not migrate a whole process context
- Transfer of *per-connection* state to another server
  - TCP state
  - Application-specific state
    - must be exported to the kernel
- Synchronization of application and TCP state

Connection State



The Application Interface

- A server application exports a per-connection *state snapshot* to the kernel
  - defines an execution *restart point*
- OS API for migrating connections
  - set a state snapshot at the origin server
  - get the state snapshot at the destination server
- Application is responsible for packing/unpacking the state

Example: Use of Migration API

S1 - origin

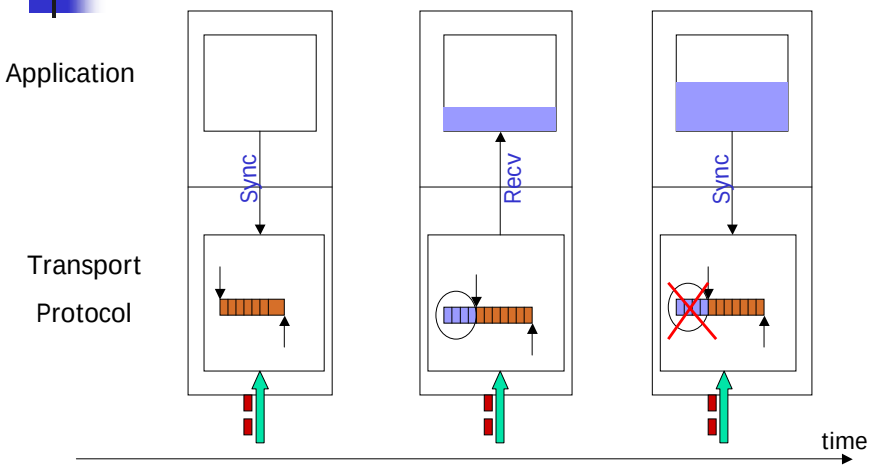
```
s = accept(ssock)
send(s, 1)
state = 1
set_state(s, &state)
....
send(s, 2)
state = 2
set_state(s, &state)
```

S2 - destination

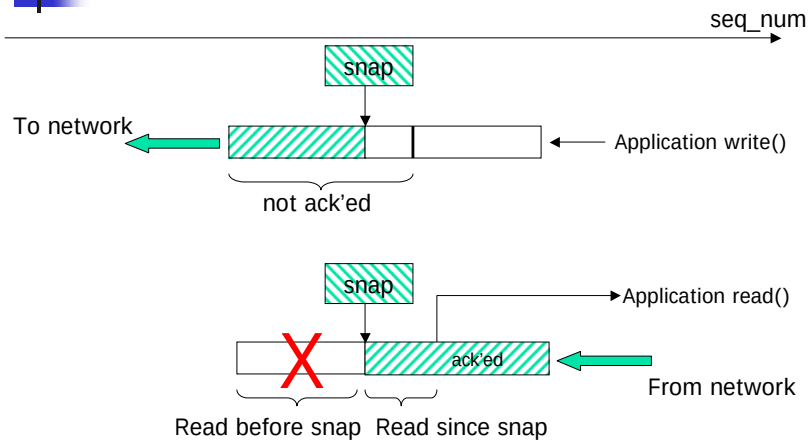
```
s = accept(ssock)
get_state(s, &state)

// determine restart point
if (state == 1)
  send(s, 2)
  ....
```

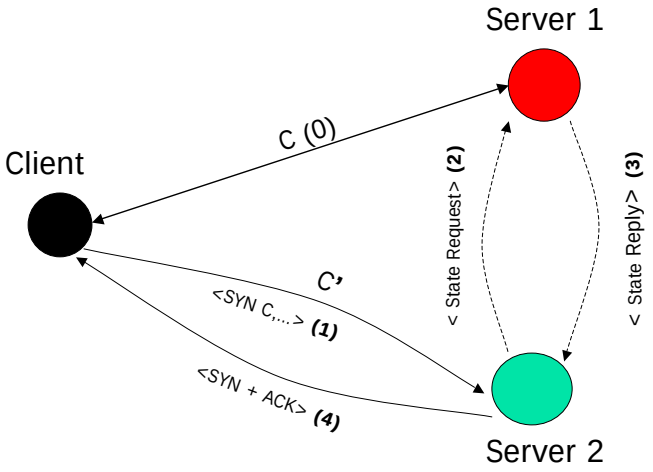

State Synchronization



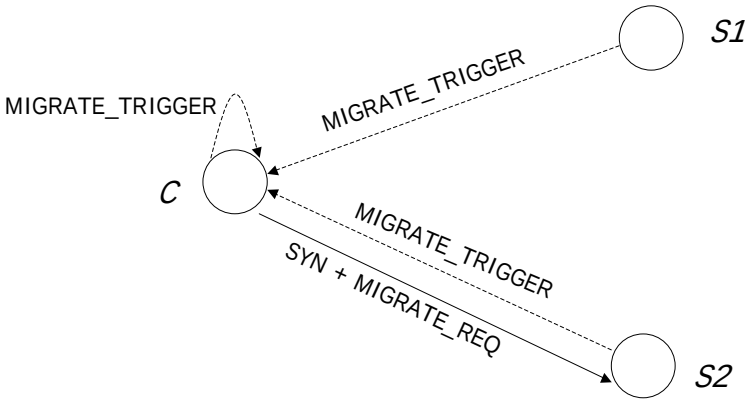
State Synchronization (cont'd)



Client-initiated Connection Transfer



Migration Architecture: Trigger (C or S) and Initiator (C)



Project Status

- Implementation of Migratory TCP in a FreeBSD kernel - almost completed
- Target applications
  - HTTP server
  - PostgreSQL front-end
- Prototype evaluation - expected in one month

Issues

- Trigger policies: when to migrate?
  - Interaction with TCP retransmission
- Connection state transfer
  - Eager vs. Lazy transfer
- Performance issues
  - When to take the state snapshot?
- Application issues
  - What application classes can the model API support ?

Summary

- Client-initiated connection migration mechanism
- Application independent, flexible
- Servers must cooperate
- Migration architecture that decouples migration triggers and policies from the mechanism

Example 1: HTTP Server

```
s = accept(ssock)
kind = get_state(s, &state)  // probe socket
if (kind == NEW_CONN) { /* new connection: start */
  // ... parse request, get file name fname
  state.fname =  fname
  state.off = 0
  set_state(s, &state)  // snap (first)
  fd = open(fname)
} else if (kind == MIGRATED_CONN) {  /* migrated connection: resume */
  fd = open(state.fname)
  lseek(fd, state.off, SEEK_SET)
}
/* both new and migrated connection */
// in a loop, send from file... and periodically save the offset
state.off = lseek(fd, 0, SEEK_CUR)
set_state(sock, &state)  // snap
```


Example 2: Transaction (S1)

```
Read(params)                    // read params from client

set_state("begin", params)      // store params

tid = BeginTransaction()
set-state("started", tid)      // needed to abort at new server, if migrated
  ...
  ...
set-state("prepare", tid)

if (PrepareCommit(tid) == OK) // assume PrepareCommit() is idempotent
  set-state("commit", tid)
  Commit(tid)
```

Example 2: Transaction (S2)

```
get-state(stage, params)      // store params

switch (stage)
  case begin:
    tid = BeginTransaction()
    ...
  case started:
    AbortTransaction(tid)
    tid = BeginTransaction()
    ...
   case prepare:
     if (PrepareCommit(tid) == OK)
     ...
   case commit:
     if (NOT Commited(tid))
          Commit(tid)
```


Example 3: Transaction at S1

```
read(s, params)                          // receive params from client
set_state(s, "trans-begin", params)      // snap: store stage & params

BeginTransaction()
  ...
  ...
status = lock_state(s)
if (status == CONN_STILL_HERE) {
  set_state(s, "trans-committed", results) // snap: store stage & results
  CommitTransaction()
  unlock_state(s)
  write(s, results)                      // send results to client
}
else if (status == CONN_MIGRATED) {
  RollbackTransaction()
  close(s)
}
```

Example 3: Transaction at S2

```
get_state(s, stage, data)      // store params

switch (stage) {
   case “trans-begin”:
      BeginTransaction()
      // use data as params for transaction
      ...
      break;

  case “trans-committed”:
      // use data as results to be sent to client
      write(s, data)
}
```