Migratory TCP and Smart Messages: Two Migration Architectures for High Availability
Network-Centric Applications

- Network Services
  - services vs. servers
  - clients expect *service availability* and quality
  - internet protocol limitations

- Massive Networks of Embedded Systems
  - dynamic networks with volatile nodes and links
  - applications to expect *result availability* and quality
  - traditional distributed computing inadequate
Internet Protocol Limitations

- **Resource Location**
  - eager mapping of resources to hosts (IP addresses)
  - mapping assumed stable and available

- **Connection-Oriented Communication**
  - reliable end-to-end communication
  - rigid end-point naming (hosts, not resources)
  - service bound to server during connection
The Distributed Computing Model

- Networks of computers with relatively stable configuration and identical nodes
- Distributed applications with message passing communication
- Deterministic execution, always returns the expected result (100% quality)
- Routing infrastructure
- Fault tolerance: node failures are exceptions
Availability Issues

- Service availability hard to achieve
  - end-to-end: server availability not enough
  - connectivity failures: switch to alternative servers
  - mobile resources may change hosts

- Result availability is even harder
  - volatile nodes: dynamic configuration
  - dynamic resources: content-based naming
  - peer-to-peer communication: no routing infrastructure
Vision and Solutions

- Relaxed Transport-Layer Protocols
  - relax end-point naming and constraints
  - \textit{Migratory TCP}: server end-point \textit{migration} for live connections

- Cooperative Computing
  - distributed computing over dynamic networks of embedded systems
  - \textit{Smart-Messages}: execution \textit{migration} with self-routing
Migratory TCP: A Relaxed Transport Protocol for Network-based Services
TCP-based Internet Services

- Adverse conditions to affect service availability
  - internetwork congestion or failure
  - servers overloaded, failed or under DoS attack
- TCP has one response
  - network delays $\Rightarrow$ packet loss $\Rightarrow$ retransmission
- TCP limitations
  - early binding of service to \textit{a} server
  - client cannot dynamically switch to another server for sustained service
Migratory TCP: At a Glance

- Migratory TCP offers another solution to network delays: connection migration to a “better” server
- Migration mechanism is *generic* (not application specific) *lightweight* (fine-grain migration of a *per-connection state*) and *low-latency* (application not on critical path)
- Requires changes to the server application but totally transparent to the client application
- Interoperates with existing TCP
The Migration Model
Architecture: Triggers and Initiators

Client

Server 1

Server 2

MIGRATE_TRIGGER

MIGRATE_TRIGGER

MIGRATE_INITIATE
Per-connection State Transfer

Server 1

Connections

Application

M-TCP

Server 2

Connections

Connections
Application- M-TCP Contract

- Server application
  - Define per-connection application state
  - During connection service, export snapshots of per-connection application state when consistent
  - Upon acceptance of a migrated connection, import per-connection state and resume service

- Migratory TCP
  - Transfer per-connection application and protocol state consistent with the last export from the old to the new server
Migration API

- `export_state(conn_id, state_snapshot)`
- `import_state(conn_id, state_snapshot)`
State Synchronization Problem
Log-Based State Synchronization

- Logs are maintained by the protocol at server
  - discarded at export_state time (state is sync’ed)
- Logs are part of the connection state to be transferred during migration
- Service resumes from the last exported state snapshot and uses logs for execution replay
Design Issues

- Robustness to server failures: when to transfer the connection state?
  - Eager vs. Lazy transfer

- Trigger policies: when to initiate connection migration?
  - policy = metric + trigger

- M-TCP overhead vs. Migration Latency:
  - When/how often to export the state snapshot?
Prototype Implementation

- Modified the TCP/IP stack in FreeBSD kernel
- Lazy connection migration
- Experimental setup
  - Two servers, one client: P II 400MHz, 128 MB RAM
  - Servers connected by dedicated network link
- Synthetic microbenchmark
- Real applications
  - PostgreSQL front-end
  - Simple streaming server
Lazy Connection Migration

Client

Server 1

Server 2

\(<\text{State Request}\>(1)\)

\(<\text{State Reply}\>(3)\)

\(<\text{SYN C, ...}\>(1)\)

\(<\text{SYN + ACK}\>(4)\)

\(C(0)\)

\(C'\)
Microbenchmark

Endpoint switching time vs. state size
Streaming Server Experiment

- Server streams data in 1 KB chunks
- Server performance degrades after sending 32 KB
  - emulated by pacing sends in the server
- Migration policy module in the client kernel
  - Metric: inbound rate (smoothed estimator)
  - Trigger: rate drops under 75% of max. observed rate
Stream Server Experiment

Trace of sequence numbers received by a client during a 256 KB download

Effective throughput close to average rate seen before server performance degrades
Protocol Utilization

- For **end-to-end availability**
  - applications with long-lived connections
  - critical applications (banking, e-commerce, etc.)

- For **load balancing**
  - migration trigger: at server side, based on load balancing policy

- For **fault tolerance**
  - eager transfer of connection state
M-TCP Limitations

- Requires TCP changes
  - use existing multi-home protocols such as SCTP
- Multiple server processes and/or connections
  - recursive state migration: hard problem
- Lazy transfer does not address server failure
  - alternative state transfer mechanism (eager, at the client)
Relaxed Transport Protocols

Autonomous Transport Protocols
- content-based end-point naming
- lazy *end-point* to *network address* binding
- apply P2P techniques to *(re)*discover the end-point location during connection

Split Transport Protocols
- split connection in the network
- involve intermediate nodes in recovery, flow and congestion control
- packet replication to tolerate intermediate node failures
Smart Messages: A Software Architecture for Cooperative Computing
Distributed Embedded Systems

- Massive ad-hoc networks of embedded systems
  - dynamic configuration
  - volatile nodes and links

- Distributed collaborative applications
  - multiple intelligent cameras “collaborate” to track a given object
  - same-model cars on a highway “collaborate” to adapt to the road conditions

- How to program and execute collaborative applications on networks of embedded systems?
  - IP addressing and routing does not work
  - traditional distributed computing does not work
Cooperative Computing

- Distributed computing through execution migration
  - Execution units: **Smart Messages**
  - Network memory: **Tag Space**

- **Smart Messages**
  - migrate through the network and execute on each hop
  - routing controlled by the application (self-routing)

- **Embedded nodes**
  - admit, execute and send smart messages
  - maintain local Tag Space
Example of a Distributed Task

Determine average temperature in town
Smart Messages (SM)

- Components
  - (mobile) code and (mobile) data bricks
  - a lightweight state of the execution

- Smart Message life cycle:
  - creation
  - migration
  - execution
  - cached code

- Distributed application: a collection of SMs
Tag Space (SM)

- Collection of named data persistent across SM executions
- SM can create, delete, read and write tags
  - protected using access rights (SM signatures)
  - limited lifetime
- I/O tags maintained by the system drivers: Temperature

<table>
<thead>
<tr>
<th>Name</th>
<th>Access</th>
<th>Lifetime</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Temperature</td>
<td>any</td>
<td>infinite</td>
<td>80</td>
</tr>
<tr>
<td>Route_to_Temp</td>
<td>{SM sign}</td>
<td>4000</td>
<td>neighbor3</td>
</tr>
</tbody>
</table>
Tag Space(SM) cont’d

- What they are used for:
  - content-based addressing: migrate \(\{\text{tag1,tag2}\}\)
  - I/O port access: read (temperature)
  - data storage: write (tag, value)
  - inter SM communication
  - synchronization on tag update: block(tag, timeout)
  - routing
SM Execution

- Non-preemptive but time bounded
- Access SM data
- Access Tag Space
- Create new SM
- Migrate
Smart Message Example 1

Tag Space

<table>
<thead>
<tr>
<th>LED Device</th>
<th>Light_switch</th>
<th>Light_status</th>
</tr>
</thead>
</table>

| Light Signal Device | Three signal |

<table>
<thead>
<tr>
<th>Smart Messages</th>
</tr>
</thead>
<tbody>
<tr>
<td>block(light_sw)</td>
</tr>
<tr>
<td>create(Three_sign) for(;;)</td>
</tr>
<tr>
<td>block(Three_sig) for (0 to 2)</td>
</tr>
<tr>
<td>write(Light_sw,1) block(Light_st)</td>
</tr>
<tr>
<td>write(Light_sw,0) block(Light_st)</td>
</tr>
</tbody>
</table>

SM 1

SM 2

write (Three_sig)
Smart Message Example 2

Tag Space

Smart Messages

SM 2
Migrate(Fire)

Fire

for(;;)
block(Image)
if (Red)
create (Fire)
Loc=read(Location)
write(Fire,Loc)

SM 1

Fire Detector

Intelligent Camera Device with GPS

Image
Location

write(Image)
Smart Message Migration

migrate (\{tag1, tag2, \ldots\}, \text{timeout})

- \{tag1, tag2, \ldots\}: content-based destination address
- \text{timeout}: abandon migration after timeout and return
- content-based routing is implemented using additional smart messages and the Tag Space

![Diagram of message migration]
Self-Routing Example (step 1)

Migrate(tag,timeout) {
    do
        if (!route_to_tag)
            create(Explore_SM)
            block(route_to_tag)
        sys_migrate(route_to_tag)
    until tag;
}

Explore_SM {
    do
        sys_migrate(all_neighbors)
        write(previous_to_tag,previous())
    while !(tag || route_to_tag)
    do
        sys_migrate(previous_to_tag)
        write(route_to_tag,previous())
    while previous_to_tag
Self-Routing Example (step 2)

Migrate(tag,timeout) {
    do
        if (!route_to_tag)
            create(Explore_SM)
            block(route_to_tag)
            sys_migrate(route_to_tag)
    until tag;
}

Explore_SM {
    do
        sys_migrate(all_neighbors)
        write(previous_to_tag,previous())
        while !(tag || route_to_tag)
        do
            sys_migrate(previous_to_tag)
            write(route_to_tag,previous())
        while previous_to_tag
    }
Self-Routing Example (step 3)

Migrate(tag,timeout) {
    do
        if (!route_to_tag)
            create(Explore_SM)
        block(route_to_tag)
        sys_migrate(route_to_tag)
    until tag;
}
Status

- Prototype implementation
  - hardware: iPAQs and Bluetooth
  - software: Java KVM and Linux

- Self-Routing
  - “pull” routing info (similar to Directed Diffusion[Estrin’99]):
  - “push” routing info (similar to SPIN[Heinzelman’99])
  - Compare their performance using a SM network simulator

- Security issues: not addressed yet
Routing Information Flooding

Simulation result
Cooperative Computing: Summary

- Distributed computing expressed in terms of computation and migration phases
- Content-based naming for target nodes
- Application-controlled routing
- Is cooperative computing a good programming model for networks of embedded systems?
In Search for a Good Metric

Quality of Result (QoR) vs. Network Adversity

![Graph showing the relationship between Quality of Result (QoR) and Network Adversity. The graph illustrates that as network adversity increases, the quality of result decreases. The ideal scenario is at the top, while the real scenario shows a decrease in quality with increasing adversity.]
Conclusions

- Two ideas to improve availability for network-centric applications
  - Relaxed transport protocols: relax end-point naming and constraints
  - Cooperative computing: distributed computing with execution migration with application-controlled routing
- Two solutions: Migratory TCP and Smart Messages