Programming Outdoor Distributed Embedded Systems

Cristian Borcea

DiscoLab - Laboratory for Network Centric Computing
Indoor Distributed Computing

- Computing is distributed for performance or fault tolerance
- Nodes are computationally equivalent
- Configuration is stable (failures are exceptions)
- Networking is robust and has acceptable delays
- Relatively easy to program
  - Message passing or shared memory
  - Names are easy to translate to network addresses
- Correct programs complete and return deterministic results
Computers Go Outdoors

Distributed object tracking over a large geographical area

Cars collaborating for a safer and more fluid traffic
Outdoor Distributed Systems

- Embedded in non-traditional computing systems
- Functionally heterogeneous
- Distributed across the physical space
- Node’s role in computation often driven by location
- Communicate through short-range wireless
- Create networks of embedded systems
  - Ad hoc topologies
How to Program Outdoor Distributed Embedded Systems?

- Recent research in networked embedded systems has focused on
  - Hardware
  - Operating Systems
  - Network Protocols
  - Data collection/dissemination in sensor networks

- Our research focuses on programmability
  - How to program distributed applications over networks of embedded systems?
Traditional Distributed Computing Does Not Work

- End-to-end data transfer may hardly complete
- Fixed address naming and routing (e.g., IP) are too rigid
- Difficult to deploy new applications in existing networks
- Outdoor distributed computing requires novel programming models and system architectures
Example

- Mobile sprinkler with temperature sensor
- Hot spot

- "Water the hottest spot on the Left Hill" 
- Number and location of mobile sprinklers are unknown
- Configuration is not stable over time
  - Sprinklers move
  - Temperature changes
Outline

- Motivation
- Spatial Programming Model
- Smart Messages System Architecture
- Implementation and Evaluation
- Conclusions
- Future Work
Programs access data through variables

Variables mapped to physical memory locations

Page Table + OS guarantees reference consistency
Application: Perform intrusion detection on Left Hill

Need a simple programming model

- Hide the networking details
- Access to embedded systems as simple as access to variables
<table>
<thead>
<tr>
<th>Virtual Address Space</th>
<th>Space Region</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variables</td>
<td>Spatial References</td>
</tr>
<tr>
<td>Variables mapped to physical memory</td>
<td>Spatial references mapped to systems embedded in the physical space</td>
</tr>
<tr>
<td>Reference consistency</td>
<td>?</td>
</tr>
<tr>
<td>Bounded access time</td>
<td>?</td>
</tr>
</tbody>
</table>
Spatial Programming (SP) at a Glance

- Program outdoor distributed applications using spatial references
- Shields programmers from networking details by providing a virtual address space over networks of embedded systems
- Embedded systems/nodes named by their expected locations and properties
Space Regions

Hill = new Space({lat, long}, radius);

- Virtual representation of a physical space
- Similar to a virtual address space in a conventional computer system
Spatial References

- Defined as \{\text{space:property}\} pairs
- Virtual names for nodes in the network
- Similar to variables in conventional programming
  - Have meaning only within the application that defined them
- Indexes used to distinguish among similar systems in the same space region
Reference Consistency

- At first access, a spatial reference is mapped to an embedded system located in the specified space.

- Mappings maintained in per-application Mapping Table (MT):
  \{space, property, index\} \rightarrow \{network_address, location\}

- Subsequent accesses to the same spatial reference must access the same system as long as it is located in the same space region (located using MT).
Reference Consistency Example

{Left_Hill:robot[0]}.move = ON;

{Left_Hill:robot[0]}.move = OFF;

Time
Space Casting

Left Hill

Right Hill

\{\text{Left\_Hill:robot[0]}\}

Left Hill

Right Hill

\{\text{Right\_Hill:(Left\_Hill:robot[0])}\}
Bounding the Access Time

- How to bound the time to access a spatial reference?
  - Discover an unmapped system for a new spatial reference
  - Mapped systems may move, go out of space, or disappear

- Solution: associate an explicit timeout with the spatial reference access

```java
try{
    {Hill:robot[0], timeout}.camera = ON;
} catch(TimeoutException e){
    // the programmer decides the next action
}
```
for(i=0; i<1000; i++)
    try{
        if ({Left_Hill:Hot[i], timeout}.temp > Max_temp)
            Max_temp = {Left_Hill:Hot[i], timeout}.temp;
        Max_id = i;
    }catch(TimeoutException e)
        break;
{Left_Hill:Hot[Max_id], timeout}.water = ON;

Application: Water the hottest spot on the Left Hill

Spatial Programming Example
Outline

- Motivation
- Spatial Programming Model
- Smart Messages System Architecture
- Implementation and Evaluation
- Conclusions
- Future Work
Smart Messages at a Glance

- **Smart Message (SM)**
  - User-defined distributed application
  - Composed of code bricks, data bricks, and execution control state
  - Executes on nodes of interest named by properties
  - Self-routes between nodes of interest

- **Self-Routing**
  - Application-level routing executed at every node
  - Applications can change routing during execution

- **Cooperative Nodes**
  - Execution environment (Virtual Machine)
  - Memory addressable by names (Tag Space)
"Dumb" vs. Smart Messages

Lunch:

Appetizer

Entree

Dessert

Data migration

Execution migration

22
Application Example

n=0
while (n< NumTaxis)
migrate(Taxi);
if (readTag(Available))
    writeTag(Available, false);
    writeTag(Location, myLocation);
n++;

need 2 taxis

n=0
n=0
n=0
n=1
n=1
n=2

data brick
application
code brick
routing
code brick
Cooperative Node Architecture

Network

Admission Manager

Code Cache

SM Ready Queue

Virtual Machine

Tag Space

SM Platform

Operating System & I/O

Network

SM
Admission

- Ensures progress for all SMs in the network
- Prevents SMs from migrating to nodes where they cannot achieve anything
- SMs specify lower bounds for resource requirements (e.g., memory, bandwidth)
- SMs accepted if the node can satisfy these requirements
  - SMs transfer only the missing code bricks
- More resources can be granted according to admission policy
Execution at a Node

- Non-preemptive, but time bounded
- Ends with a migration, or terminates
- During execution, SMs can
  - Spawn new SMs
  - Create new SMs out of their code and data bricks
  - Access the tag space
  - Block on a tag to be updated
Tag Space

- **Application tags**: “persistent” memory for a limited duration across SM executions
- **I/O tags**: uniform interface for interaction with operating system and I/O subsystem
- **Tags are used for**
  - Content-based naming: `migrate(tag, timeout)`
  - Inter-SM communication: `write(tag, data), read(tag)`
  - Synchronization: `block(tag, timeout)`
  - I/O access: `read(temperature)`
- **5 protection domains for access control**
- **migrate()**
  - implements routing algorithm
  - migrates application to next node of interest
  - names nodes in terms of arbitrary conditions on tag names and tag values

- **sys_migrate()**
  - one hop migration
Routing Example

RouteToTaxi = 2

RouteToTaxi = j

migrate(Taxi){
    while(!readTag(Taxi))
        if (readTag(RouteToTaxi))
            sys_migrate(readTag(RouteToTaxi));
        else
            create_SM(DiscoverySM, Taxi);
            createTag(RouteToTaxi, lifetime, null);
            block_SM(RouteToTaxi, timeout);
}
Self-Routing

- **SMs carry the routing and execute it at each node**
- **SMs control their routing**
  - *Select routing algorithm (migrate primitive)*
    - Multiple library implementations
    - Implement a new one
  - *Change routing algorithm during execution in response to*
    - Adverse network conditions
    - Application’s requirements
Example of Dynamic Change of Routing

SM starts with routing for dense networks

migrate(Taxi, timeout1)

Dense network

Sparse network

migrate timeouts

SM continues with routing for sparse networks

migrate(Taxi, timeout2)
Self-Routing Simulation Results

On-Demand Routing versus Geographic + On-Demand Routing

- 3 nodes of interest located in the corners have to be visited in clockwise order
- vary the radius from 100m to 700m

 Starting node  Node of interest  Other node

On-Demand Routing versus Geographic + On-Demand Routing

<table>
<thead>
<tr>
<th>Region Radius (meters)</th>
<th>Completion Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>100</td>
<td>1.2</td>
</tr>
<tr>
<td>200</td>
<td>2.4</td>
</tr>
<tr>
<td>300</td>
<td>3.6</td>
</tr>
<tr>
<td>400</td>
<td>4.8</td>
</tr>
<tr>
<td>500</td>
<td>6.0</td>
</tr>
<tr>
<td>600</td>
<td>7.2</td>
</tr>
<tr>
<td>700</td>
<td>8.4</td>
</tr>
<tr>
<td>800</td>
<td>9.6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Region Radius (meters)</th>
<th>Bytes Sent in the Network (KBytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>200</td>
<td>200</td>
</tr>
<tr>
<td>300</td>
<td>300</td>
</tr>
<tr>
<td>400</td>
<td>400</td>
</tr>
<tr>
<td>500</td>
<td>500</td>
</tr>
<tr>
<td>600</td>
<td>600</td>
</tr>
<tr>
<td>700</td>
<td>700</td>
</tr>
<tr>
<td>800</td>
<td>800</td>
</tr>
</tbody>
</table>
Prototype Implementation

- Modified version of Sun’s Java K Virtual Machine
  - Small memory footprint (160KB)
- SM and tag space primitives implemented inside virtual machine as native methods (efficiency)
- Implemented I/O tags: GPS location, neighbor discovery, image capture, light sensor, system status

Prototype Node with GPS receiver and video camera
Lightweight Migration

- Traditional process migration difficult
  - Strong coupling between execution entity and host
  - Needs to take care of operating system state (e.g., open sockets, file descriptors)

- Tag space decouples the SM execution state from the operating system state

- SM migration transfers only
  - Data bricks explicitly specified by programmer
  - Minimal execution control state required to resume the SM at the next instruction (e.g., instruction pointer, operand stack pointer)
Experimental Results for Simple Routing Algorithms

HP iPAQs running Linux and using IEEE 802.11 for wireless communication

<table>
<thead>
<tr>
<th>Routing algorithm</th>
<th>Code not cached (ms)</th>
<th>Code cached (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Geographic</td>
<td>415.6</td>
<td>126.6</td>
</tr>
<tr>
<td>On-demand</td>
<td>506.6</td>
<td>314.7</td>
</tr>
</tbody>
</table>
Spatial Programming Implementation Using Smart Messages

- SP application translates into an SM
- Spatial reference access translates into an SM migration to the mapped node
- Embedded system properties: Tags
- SM self-routing (content-based and geographical routing)
- Reference consistency
  - Unique tag created when a spatial reference is mapped to a node
  - Name of the unique tag and the location of the node stored in Mapping Table (MT)
Max_temp = \{\text{Left_Hill}:\text{Hot}[1], \text{timeout}\}.\text{temp};

\{\text{Left_Hill},\text{Hot},1\} \rightarrow \{\text{yU78GH5},\text{location}\}

\text{ret} = \text{migrate\_geo(location, timeout)};
\text{if} \ \text{ret} == \text{LocationUnreachable}
\text{ret} = \text{migrate\_tag(yU78GH5, timeout)};
\text{if} \ (\text{ret} == \text{OK}) \ && (\text{location} == \text{Left\_Hill})
\text{return readTag(temp)};
\text{else} \ \text{throw TimeoutException}
SP Application: Intrusion Detection

10 HP iPAQs with 802.11 cards and GPS devices

- user node
- light sensor
- camera node
- regular node

monitored space

Code Size breakdown for SM
(Application + SP Library)

- Library Code: 87%
- Application Code: 11%
- Stack and Data: 2%
- Total size = 18.5KB

Code Size breakdown for SP library

- SM Wrapper: 27%
- Geographical Routing: 15%
- Auxiliary Classes: 15%
- Binding Table: 10%
- Content Based Routing: 33%
- Total Size = 15.9KB
Execution Time Breakdown

![Bar chart showing execution time breakdown for different tasks]

- **Code uncached**
  - Migrate back to user
  - Migrate to camera
  - Discover camera
  - Migrate to light sensor
  - Discover light sensor
  - Reach space

- **Code cached**
  - Migrate back to user
  - Migrate to camera
  - Discover camera
  - Migrate to light sensor
  - Discover light sensor
  - Reach space
Conclusions

- Spatial Programming makes outdoor distributed applications simple to program
- Volatility, mobility, configuration dynamics, ad-hoc networking are hidden from programmer
- Implementation on top of a Smart Messages
  - Easy to deploy new applications in the network
  - Quick adaptation to highly dynamic network configurations
Future Work: Real Applications

- **EZCab**: An automatic system for booking cabs in cities
- **TrafficView**: A scalable traffic monitoring system
- **Smiles**: SmartPhones for interacting with local embedded systems
Thank you!

http://discolab.rutgers.edu

Outdoor Distributed Computing Project People:

Professor Liviu Iftode

Graduate Students: Cristian Borcea, Porlin Kang, Nishkam Ravi, Peng Zhou

Collaboration with Professor Ulrich Kremer and Professor Chalermek Intanagonwiwat
Protection Domains for Tag Space

- **Owner**: SM that creates the tag
- **Family**: all SMs having a common ancestor with the SM that owns the tag
- **Origin**: all SMs created on the same node as the family originator of the tag owner
- **Code**: all SMs carrying a specific code brick
- **Others**: all the other SMs
Access Control Example
(Code-based protection domain)

Owner = SM1
[Hash(Cr), RW]

Cr     Same routing used by SM1 and SM2
Access permission granted for SM2
Access permission denied for SM3