

AmbientTalk with Guards

Justin Collins
CS 239
Spring 2008

What is AmbientTalk?

- Language intended for mobile ad hoc networks
- Object oriented
- Dynamically typed
- Asynchronous event handlers
- Remote method calls
- Futures
- AmbientReferences



<http://prog.vub.ac.be/amop/>

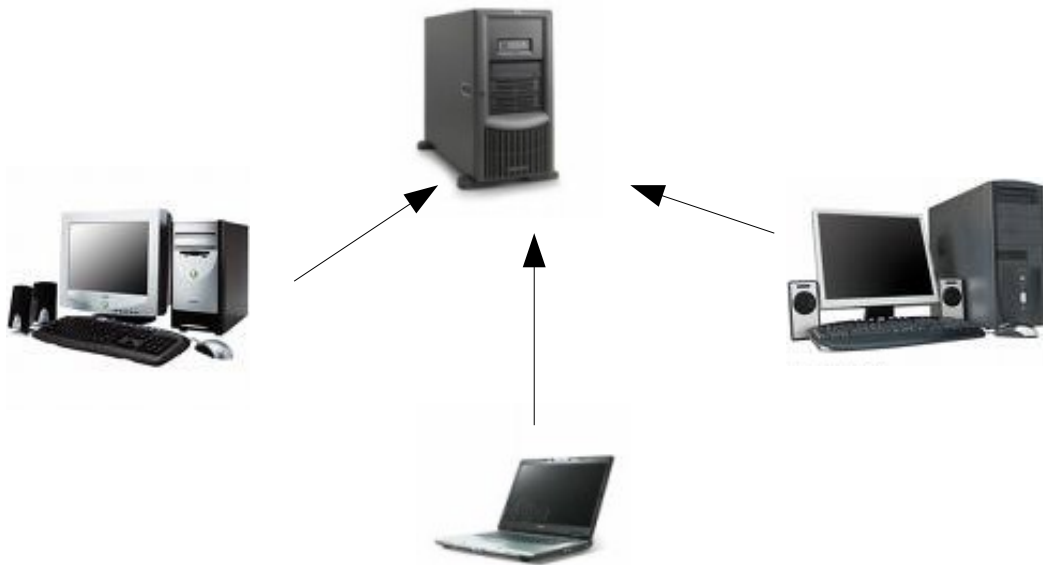
What is a MANET?

- Mobile devices
 - Cell phones, smart phones, PDAs
 - Laptops, sensors, game systems
- Ad hoc
 - No infrastructure
 - Self organizing
- Wireless Networks
 - WiFi
 - Bluetooth



Traditional Approach

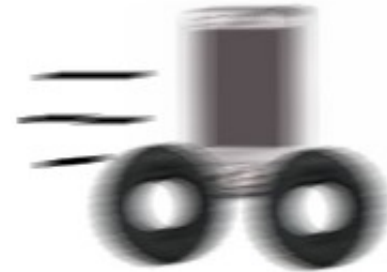
- Client-Server architecture



- Disconnection is an “exceptional” event
- Fairly stable topology
- Reliable centralized server

Issues in MANET

- Frequent disconnections
 - Mobility
 - Wireless channel
- Dynamic topology
 - Nodes join and leave often
 - No fixed infrastructure
 - Transient connections
- Small devices
 - Limited power, battery life



What Does AmbientTalk Provide?

//Once

when: SomeType discovered: { };

//Every time until canceled

whenever: SomeType discovered: { };

when: SomeObject disconnected: { };

when: SomeObject reconnected: { };

What Might We Want?

when: SomeType **discovered:** { }

if: { stars == "aligned" };

whenever: SomeType **discovered:** { }

if: { buffer.size < max_buffer };

First Thought

- Use OMeta
 - Create parser
 - Add cool language extensions
 - “Compile” back to regular AmbientTalk
- Result
 - Pretty much working parser

<http://jarrett.cs.ucla.edu/ometa-js/#JCTry>
- But
 - That was probably overkill
 - AmbientTalk is easy to extend

Second Thought

- Just extend AmbientTalk a bit
- It is easy to add/override built-ins
- Take advantage of keyworded methods

```
def whenever: SomeType discovered:  
  code  
  if: guard;
```

Implementation

- Wrap regular **whenever: ... discovered:**
...
- User timer to check conditions
- Replace “boring” subscription object with:

FlexibleSubscription {

```
def pause();  
def resume();  
def cancel();  
def restart();  
def manual_check();
```

```
}
```

But All You Really Need to Do...

```
whenever: SomeType discovered: { |st|  
  do_something_awesome(st);  
} if: { buffer.size < max_buffer };
```

Which will

- Execute code whenever SomeType found and condition met
- Periodically check condition against previously discovered SomeTypes

Example

- Networked game (let's say Hearts)
- Requires exactly four (4) players



Player Two



Player One



Player Three



Player Four

Example - Core Code

```
whenever: GamePlayer discovered: { |player|  
  when: player<-get_name() becomes: { |name|  
    system.println(name + " has joined the game.");  
    players << player;  
    if: (paused.and: { players.length == min_players } ) then: {  
      resume();  
    };  
    when: player disconnected: {  
      system.println(name + " has left the game.");  
      players.remove(player);  
      pause();  
    };  
  };  
} if: {  
  players.length < min_players;  
};
```

What It Does

- If there are less than four players
 - Pauses game
 - Waits to discover new players
- When new player found
 - Adds player to game
 - Resumes game if there are four players
- When a player disconnects
 - Pauses game
 - Waits to discover new players

Work To Do

- Better condition checking - reactive not polling
- Add more guards where useful
- Add more shortcuts?
 - **whenever:** Type **discovered:** {} **until:** { };
 - **whenever:** Type **exactly:** Num **discovered:** {};
 - **select:** Type **where:** {} **discovered:** {};