Embedded Software Engineering Course
Winter 2005/2006
reMOTEable Project

Alois Hofstätter, Bernhard Kast, Horst Stadler

February 10, 2006

Contents

1 Introduction

1.1 Our Hardware

5 pieces of Telos mote IV Rev A (2k RAM, 60k ROM)

Figure 1: Telos mote IV

1.2 Existing Solutions

- Agilla
  - Existing agent system
  - Strong mobility
  - Too high hardware requirements
- Mate
  - Not an agent system
  - A mote reprogramming system
- Sensorware
  - Too high hardware requirements
  - Weak mobility

⇒ These solutions are not suitable, we need our own system

1.3 Our Goal

Our system will be an agent system with multiple agents in one network and it should also provide strong mobility, this means it should not fail if changes in the network structure happen. The motes should be able to communicate with each other and if needed with PCs too. We plan to inject a new agent into a running system via the PC. The final system should be able to handle data collection agents (like sensor networks).

1.4 TinyOS

TinyOS supports our hardware very well and we use its functionality. TinyOS is event based and organized as a layered system. The functionalities are divided into components an can be combined as needed. Those components communicate with each other using events and commands. TinyOS uses a FIFO scheduling schema. Task can be interrupted by events or commands but not by other tasks.Each task as to be finished before another task can be started. The programming language is called nesC and is similar to the well known C language. nesC does not support dynamic memory allocation nor function pointers. We used the TinyOS version 1.x.

2 Design thoughts

To get an flexible, hardware and OS independent agent system we decided to design our own language and therefor we needed to implement a virtual machine. Even we would restrict the agent system to TinyOS platforms a VM would be necessary anyway because TinyOS does not provide dynamic memory allocation or function pointers. The agents should be given the ability to jump from one mote to the next by a radio interface. In fact we had to design and implement some kind of routing algorithm.

2.1 Virtual Machine

We designed the language and its architecture with respect to save memory. So the memory is divided into 4 bit blocks (half-byte alignment) and each block can be addressed. The length of the commands is variable, but is always a multiple of 4 bit even if some bits are unused. We decided to use 4 bit for command number and optional the following for arguments. To break the limitations of a small number of commands we classified the commands into basic ones and high level commands. High level commands can be called with the capability command (capcall). This is used to allow a agent to be sent over the radio interface, to read sensors or to switch leds. The capcall has two arguments, which specifies a package number (4 bit) and its function (4 bit). Functions which have similar functionality are grouped together into packages. For example all functions which have something to do with the leds are in package number 15. The available commands are listed in Table 2.

2.1.1 Architecture

The virtual machine provides four multi purpose registers, four flags to indicate a division by zero, an overflow, a remainder flag for divisions and a compare flag the allow branches. All these information always stays with the agent.

Type	Capacity	Addressing size
Register	8 bit	2 bit
Address (DS,CS)	4 bit	8 bit
Value	8 bit	

Table 1: Architecture types and their sizes

2.1.2 Code- and Datasegment partitioning

The maximum size of the CS and DS altogether is 416 Half-Bytes, because the maximum size of an agent is 432 Half-Bytes and 16 Half-Bytes are needed for the registers, the program counter and the flags. The 432 Half-Byte limit results from the maximum radio packet payload size (54 Half-Bytes) and the maximum sequence number (3 bit → 8 values) of radio transport packet. We decided that this limits are sufficient for a embedded agent system. However the size of the CS or DS can't exceed the size of 256 Half-Bytes, because the CS and DS size counter has a limited length of 8 bit (→ 256 values). See Figure 2

Nr	Name	Parameter 1	Parameter 2	Description	Length(bit)
0	xor	register 1	register 2	bitwise XOR; register 1 = register 1 XOR register 2	4 + 2 + 2
1	and	register 1	register 2	bitwise AND; register 1 = register 1 AND register 2	4 + 2 + 2
2	or	register 1	register 2	bitwise OR; register 1 = register 1 OR register 2	4 + 2 + 2
3	cmp	register 1	register 2	set compare-flag 1 if register1 == register2	4 + 2 + 2
4	jmpt	(absolute) address		Jump true	4 + 8
5	jmpf	(absolute) address		Jump false	4 + 8
6	add	register 1	register 2	register 1 = register 1 + register 2; note can be used as "mov reg1,reg2"	4 + 2 + 2
7	sub	register 1	register 2	register 1 = register 1 - register 2; note can be used to set register to 0	4 + 2 + 2
8	div	register	register	register = register / register; rest will be saved on a specific position $\Longrightarrow$ mod is also provided	4 + 2 + 2
9	mul	register	register	register = register * register	4 + 2 + 2
10	mov	register	value (byte)	register = value	4 + (2) + 2 + 8
11	load	register	address	register = address	4 + (2) + 2 + 8
12	store	address	register	address = register	4 + 8 + (2) + 2
13	capcall	value 1	value 2	call method(value 2) from capability(value 1)	4 + 4 + 4
14	<reserved>			reserved for future use	
15	exit	<none>	<none>	exit agent	4

Table 2: Commands of the Virtual Machine

Package number	Function number	Description
0		Radio link
	0	send agent to mote in register 0
	1	send agent to all neighbour motes
15		Leds
	0	toggle red led
	1	toggle green led
	2	toggle blue led
	3	switch red led on
	4	switch green led on
	5	switch blue led on
	6	switch red led off
	7	switch green led off
	8	switch blue led off
	9	show value of register 0 with the leds

Table 3: Capcalls

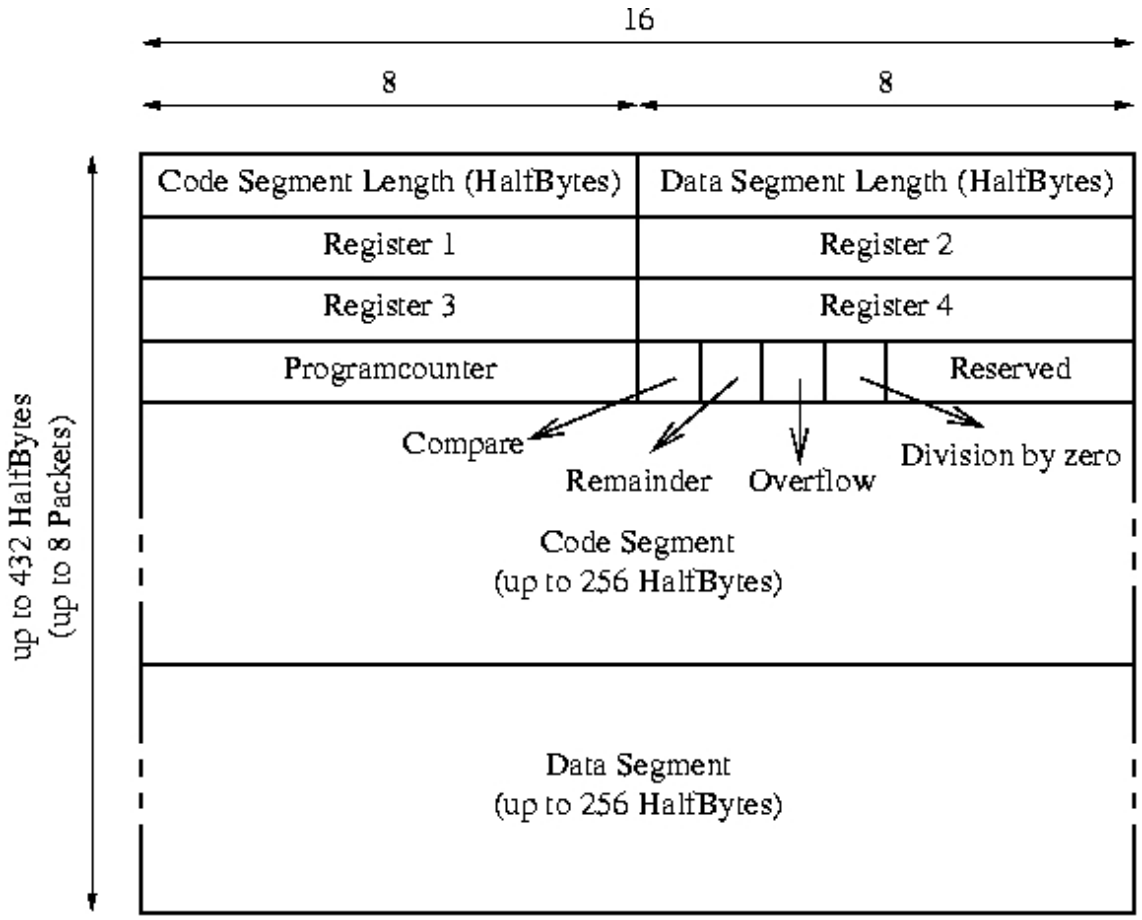


Figure 2: Agent memory layout

2.1.3 Slots and Scheduling

At this state of our implementation we support to run four agents "at the same time". Each agent has its own Virtual Machine slot. The number of simultaneously running agents is restricted by the RAM size of our motes. The scheduling is just a simple Round Robin algorithm. Each round executes one command of each agent.

2.1.4 Translator

To ease our work we created a tool for translating human readable assembler code into reMOTEable VM code. This tool was written in Java 1.5 and has a graphical user interface and a command line interface. The command line tool requires a input file and writes the generated statements to an output file. The graphical tool can use an input file or can be used like an text editor. This tool is very very helpful especially in respect to jump labels. For an example see Figure 3 and Figure 4 show a screen shot of the graphical tool.

2.2 Agent

The agent consists of a code segment and its size. The code segment is readonly and can't be accessed by the agent. Moreover the agent provides a data segment and its size. This segment can be read and written by the agent. The flags are set by the related commands. The program counter is increased by each command, except the two jump commands, which can set the program counter to the absolute given address. Most of the basic commands and most of the capability calls operate with the four registers. The mote to mote communication is provided via capability calls to the agent. For routing concerns the agent

```
mov 0 0
mov 2 1
:start1
mov 1 10
:start2
sub 1 2
capcall 15 1
cmp 1 0
jmpf start2
capcall 15 2
cmp 0 0
jmpt start1
capcall 15 0
exit
```

Figure 3: Short agent code example

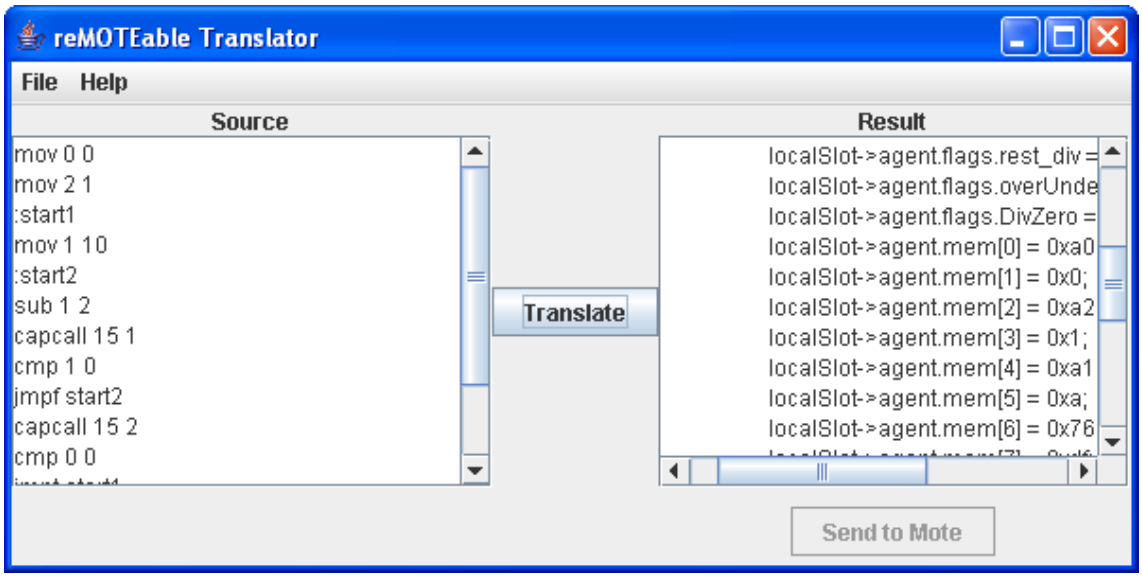


Figure 4: Screenshot of the Translator GUI

itself has to worry about. If an agent wants to get to a specific mote, which can be specified by its address or special capabilities, it can find the route itself (provide an own routing algorithm) or it can use a route acquiring agent, which is implemented on each mote. This route acquiring agent tries to find the needed destination and can be activated by a capability call.

2.2.1 Route Acquiring Agent

To get the route to the destination the route acquiring agent (RAA) first sends itself to all neighbours of the current mote and saves the origin mote address in its path list in the data segment. In the next step the RAA sends itself to all neighbours of the current mote, except to those in its path list. As before the address of the current mote is added to the path list. This step is repeated till the destinations is found or the RAA has already visited all current neighbour motes. The agents which reach the destination travel back to the origin mote using the collected path and provides the path to the calling agent. This algorithm finds all existing routes. If a RAA has no further unvisited motes (it found a dead end or a loop) it will terminate. We are aware of the issue, that this method evolves much traffic if all motes are in the same communication zone. We didn't find a satisfying solutions with agents for this problem till now.

2.2.2 Data Collecting Agent

This agent would be a possible application for sensoring networks. Suppose that some motes are spread over an area to collect data every minute. If the memory of the agent is full it needs to transfer the data to a master, which may be connected to a database. At this point the agent creates a copy of itself with an free memory. So the agent can send itself to the master, including the collected data and stay on the mote collecting further data.

2.3 Transport Protocol and Routing

In this section the necessary technics for the mote to mote communication and the routing is described.

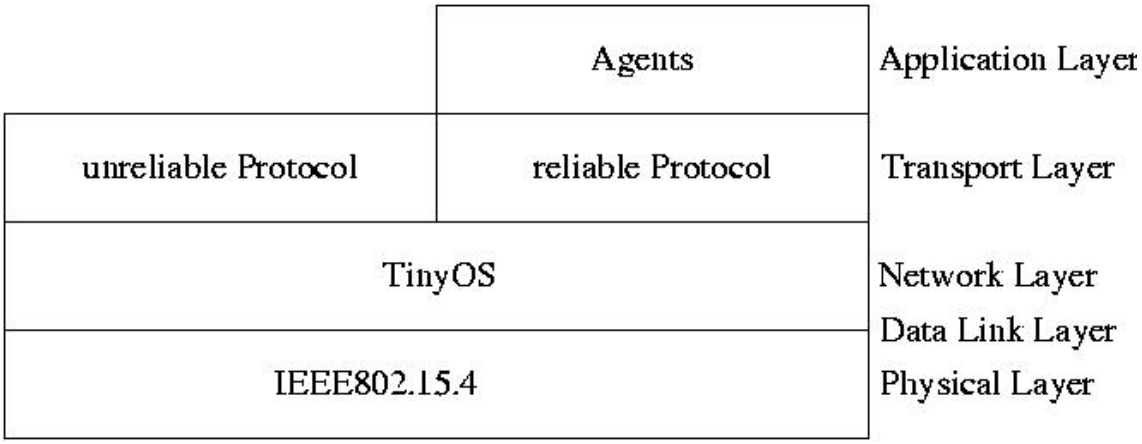


Figure 5: Network Model

2.3.1 Neighbour Discovery Protocol

If an agent wants send itself to another mote the actual mote has to know which motes are in its communication area. Therefor we designed a neighbour discovery protocol. The main concept is based on request and answer strategy. To discover its neighbours a mote sends an discovery request and all motes, which receive this broadcast request answer with an unicast response. The sender of each response is written to a local neigbours list and a corresponding time-to-life value is set. A timer decrements this time-to-life value to zero. If a value reaches zero the entry is obsolete. The reason for this design is to have caching, because periodic alive signals would be redundant and would stress the already limited resources.

A discovery protocol request packet can be distinguished from other packets, by it's destination address (broadcast address).

Name	bits
Source	4
Destination	4

Table 4: Neighbour discovery packet

2.3.2 Agent Transport Protocol

This protocol is reliable, unicast and used to send an agent from one mote to another within the communcation zone. It contains no routing information. If an agent should travel along a route, the route information is contained in the agent and it has to care about the routing itself. The discovery protocol packets can be distinguished from a transport protocol packet by it's length (Transport Protocol 2 Byte; Discovery Protocol = 1 Byte). TinyOS allows a maximum packet size of 58 Half-Bytes. 4 Half-Bytes are used for the header (see Figure 6) so we can transfer 54 Half-Bytes of an agent per packet. The session ID

and the source address are taken into account to assign an arriving packet to the correct virtual machine slot. So it's possible to transfer more than one agent form a mote A to another mote B at the same time. To transport agents with a size bigger than the payload of one packet a sequence number is used. To identifiy the last packet a "last packet bit" was introduced. Due to the reliability of the transport protocol, acknowledgments are needed. An acknowledgment confirms the arrival of the package and free resources (slots) to run the agent on the receiver mote. The ACK also carries the address of the communication partner, the sequence number and the session ID to assign this ACK to the sent packet. Both motes need a timeout. The sender needs it to retransmit a data packet. After a small number of retransmissions the whole transfer is cancelled. The receiver uses the timeout to free the allocated slot in case of a broken communication .

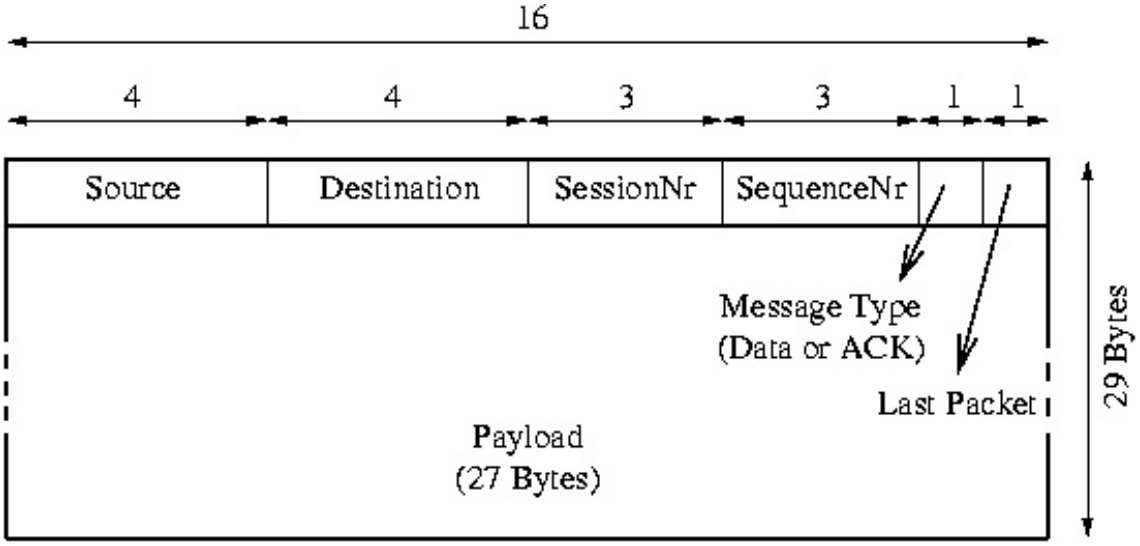


Figure 6: Transport protocol packet

3 Further Ideas

3.1 USB communication

Via the Java communication API (javax.comm) a connection between a mote and a PC can be established. This can be used to collect data from a mote / mote network or to inject new agents into the running virtual machine. The reMOTEable translator can be enhanced to allow direct uploads of the agent code to the mote.

3.2 reFODESY

As a proof of concept the "Fog Detection System" (FoDeSy) project of the past ESE course could be reimplemented with this technique.

3.3 Routing algorithm

The used routing algorithm is very simple, but has some drawbacks. So it would be interesting to find and experiment with other algorithms, which are useable for our agent concept.

4 Implementation Status

4.1 Virtual Machine

- Basic commands: implemented and working

- CapCalls:
  - The “Leds”, “Neighbour Discovery”, “Send to one neighbour mote” CapCalls are working
  - “Sensors” , “Send to all neighbour motes” CapCalls are not fully implemented or buggy
- Neighbour Discovery Protocol
  - Request works
  - Response works
  - Caching works
- Transport Protocol
  - Send part works
  - ACK part works
- Scheduling works
- Timeouts for receiving and sending

4.2 Agents

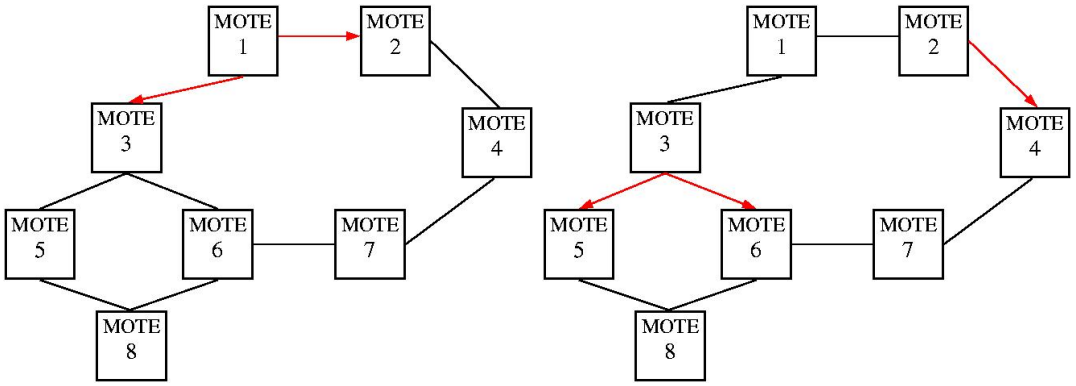
- Various test agents
- Self-Sending agent
- RAA (incomplete CapCall)

4.3 CapCall “Send to all neighbour motes”

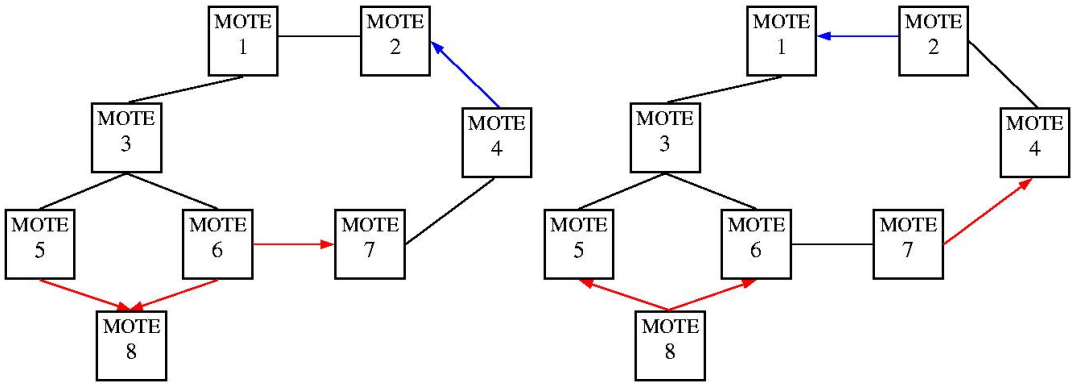
If an agent wants to be sent to more than one destinations at the same time, it will be done by the mote as followed. The mote will send the agent to the first mote and it will continue sending the agent to the second mote after the agent was successfully sent to the first mote. We did it this way to avoid high data usage for the ACK management. This can easily be realized by using the “Send to one neighbour mote” CapCall.

A Route Acquiring Agent Example

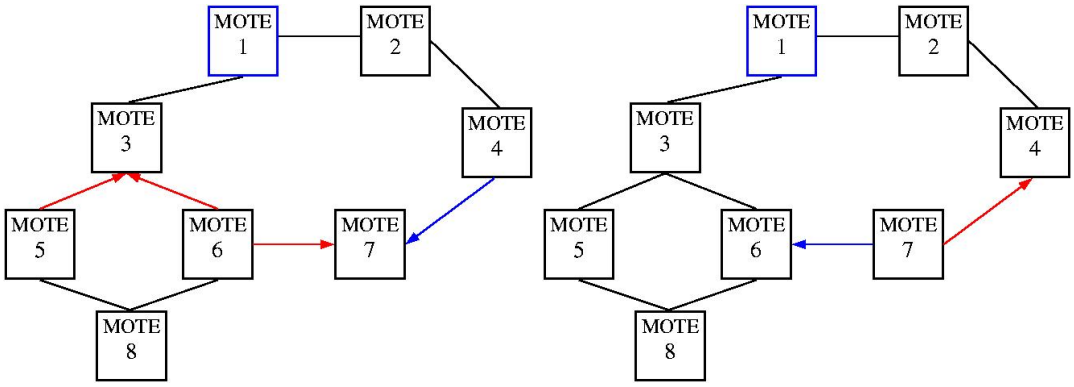
Suppose mote 1 sends a RAA to find mote 4. The red lines represent a searching RAA and the blue lines represent a successfull RAA on its way back.



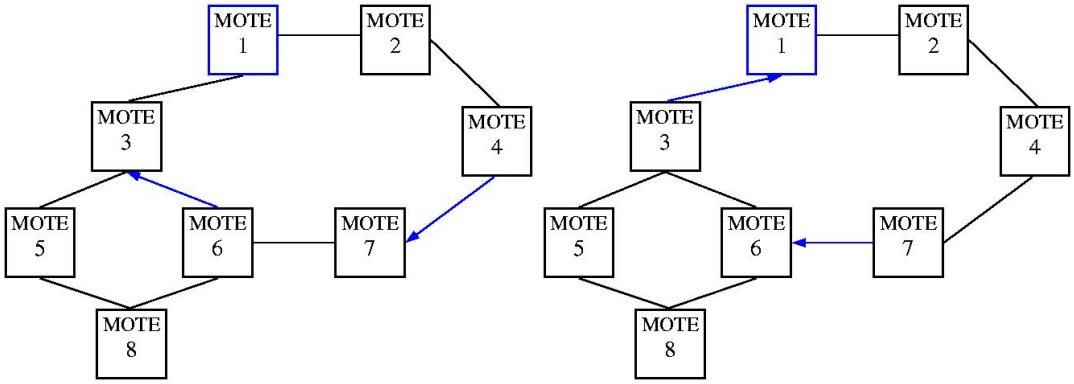
Step 1 (left) and Step 2 (right)



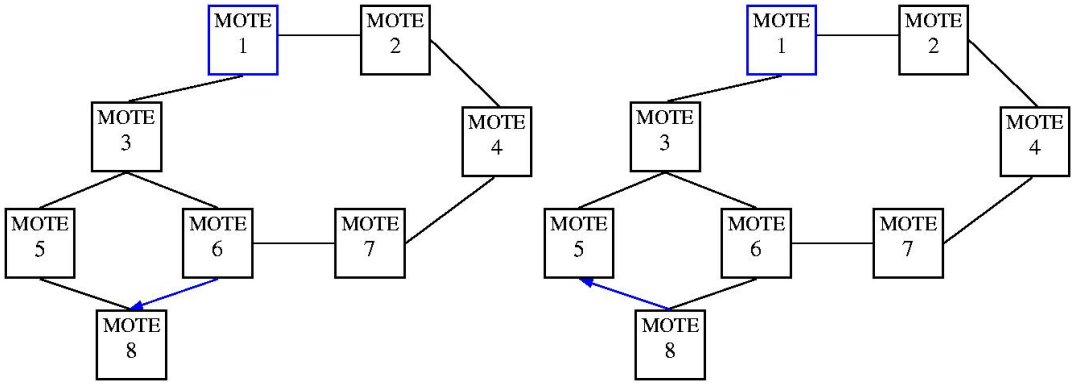
Step 3 (left) and Step 4 (right)



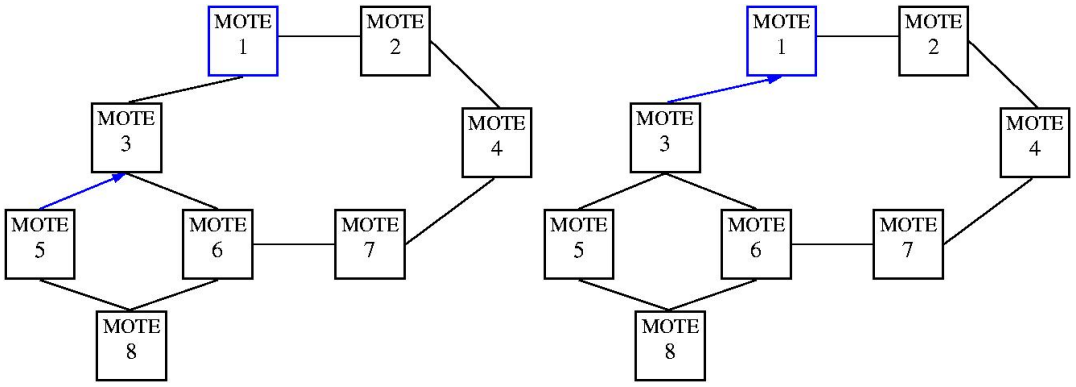
Step 5 (left) and Step 6 (right)



Step 7 (left) and Step 8 (right)



Step 9 (left) and Step 10 (right)



Step 11 (left) and Step 12 (right)

B References

Philip Levis, David Culler. Mate: A Tiny Virtual Machine for Sensor Networks.
http://www.cs.wustl.edu/l̃u/cs537s/Papers/mate.pdf

Chien-Liang Fok, Gruia-Catalin Roman, Chenyang Lu.
"Rapid Development and Flexible Deployment of Adaptive Wireless Sensor Network Applications" In Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'05), Columbus, Ohio, June 6-10, 2005, pp. 653-662.
http://www.cs.wustl.edu/mobilab/projects/agilla/index.html

http://www.moteiv.com/products-tmotesky.php

http://www.tinyos.net/