SPECIFICATION

Code mobility in Networked Embedded System

Release 0.0.1

Group 4

November 26, 2012

Igor Pelesić, Matrikelnumber 0006828
Konstantin Selyunin, Matrikelnumber 1228206
Miljenko Jakovljević, Matrikelnumber 0426673

Contents

1 Introduction

1.1 Purpose

The specification defines the goals that should meet the project "Code mobility in Networked Embedded system". It is written by project members listed on the title page to precisely identify the goals to meet at the end of the project.
The purpose of the project is to build code mobility platform on Embedded system engineering board [**?**]. Our goal is to develop the system that allows users to build and execute simple agent program on top of hardware ESE platform. To achieve the goal we propose to develop three layered structure: agent layer, platform layer, communication layer. Developer of the agent program should not aware of location of the services available on the platform and details about method to access the particular service. Given list of services agent is executed on a platform that is

1.2 Definitions, Acronyms, and Abbreviations

By *code mobility* we mean the capability of code to change the location where it is executed.

Strong code mobility is the ability to allow migration of both code and execution state to the destination, *weak* code mobility allows code transfer but it does not involve the transfer of the execution state.

Platform is a component that provides corresponding hardware services to

1.3 Background

The research has been done to use code mobility in distributed environment [**?**] and various application has been developed including [**?**] web application platform that allows people without major programming experience to develop the application as work-flow specification in graphical form. The use of code mobility is to "move the knowledge close to the resources" [**?**] and enable higher flexibility of accessing remote resources.

2 Requirements

Define requirements for the project "Code mobility in Networked Embedded system"

1 **User roles**

1.1 Application Developers (Tasks: Create control application in agent language, debug, test, prepare deployment packages)

1.2 Application Consumers (Tasks: Deploy control application on target system, fill valuable bug reports)

1.3 Plattform Developers (Tasks: Maintenance, Extensions, Porting to another target board)

1.4 Maybe: Application Designers (Tasks: Design control application)

Global Requirements:

2 **Application development requirements:**

2.1 The App Developer should be enabled to instantiate up to 4 agents on a single node, which are running concurrently.

2.2 The App Developer should be allowed to configure the execution scheduling of the agents via a prioriatization of the agents.

2.3 The platform should provide a simple agent programming language to the App Developer in which the agents of an application can be developed.

2.4 The agent language should provide the App Developer with the possibility to reproduce its code on another node or on another board.

2.5 The agent language should provide the App Developer with the possibility to communicate with another agent on the same board.

2.6 The agent language should provide the App Developer with the possibility to access the node hardware.

2.7 The agent language should provide the App Developer with the possibility to implement loops.

2.8 The agent language should provide the App Developer with the possibility to compare variables.

2.9 The agent language should provide the App Developer with the possibility to perform addition, subtraction, multiplication and division on variables.

2.10 The agent language should provide the App Developer with the possibility to perform delays in the execution of code.

2.11 The platform should allow debugging of agents executions.

2.12 The platform should provide means for the creation of easily installable deployment packages.

3 **Application Consumers requirements:**

3.1 The platform should provide means to deploy the agent software on the target boards easily.

3.2 A tracing mechanism should be provided in order to ease the process of fault detection and to allow valuable bug descriptions.

4 **Non-functional requirements**

4.1 The platform should be open to extensions i.e adding new hardware.

4.2 The agent language should be extendable.

4.3 Scalability

4.4 Documentation

4.5 A platform tracing mechanism should be provided which allows for more efficient bugfixing.

5 **What do the Application Designers need:**

5.1 A description of the platform possibilities and limitations should be provided.

5.2 The platform should provide means for reducing the overall complexity of a system, by allowing encapsulation of different tasks.

5.3 The platform should provide configurable inter agent communication facilities.

5.4 The platform should provide means to enable standby scenarios by allowing dynamical code reproduction.

5.5 The platform should provide means for strong mobility, where an agent and its execution state are transferred to a new node or board and the execution on the new destination is started from the memorized state.

5.6 A description of a platform should provide a list of all available services

3 Specification

3.1 General

The following figure depicts the general outline of the code mobility project.

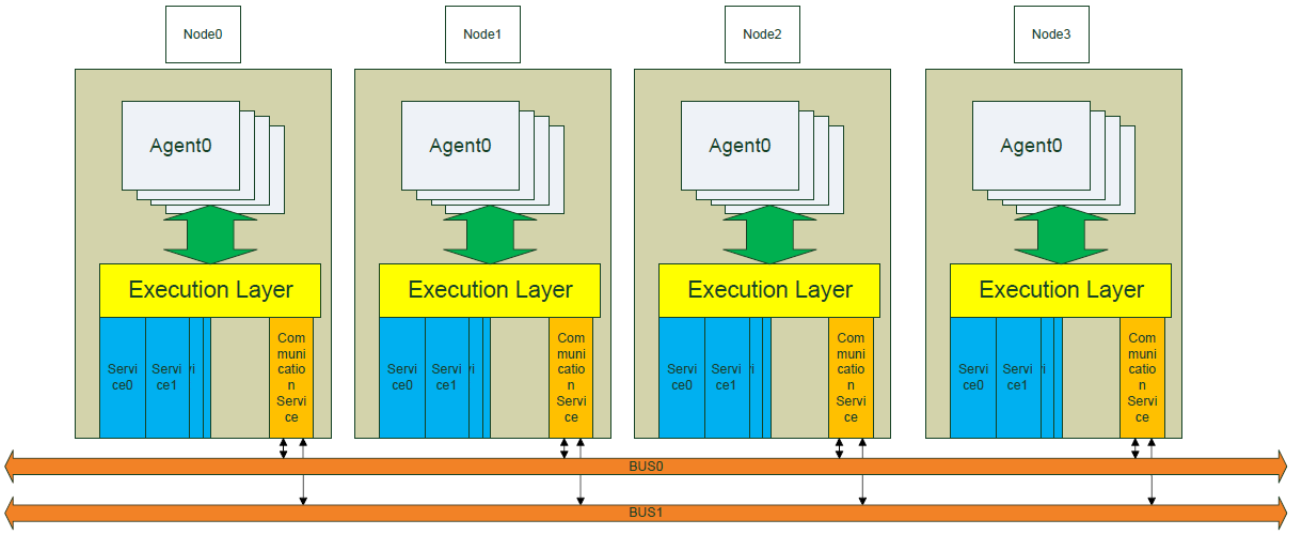

Figure 3.1: Overview

On each of the 4 nodes, which can be found the ESE board, a virtualization platform will be deployed. This virtualization platform will be able to execute up to 4 agents concurrently. The agents will be programmed in a simplistic assembler like agent language. On the platform there will be an execution layer which is able to execute the agent language. The agents will be able to access the hardware attached to a node via services which are provided by the virtualization platform. Additionally the agents can reproduce themselves to another node or even board. Within the platform a scheduler will be responsible for providing execution time to each of the agents according to their priority.

3.2 Virtualization Platform

The main task of the virtualization platform is to interprete the agent language commands of the agents and to provide them access to the hardware attached to a node via well defined interfaces. Additionally the platform should allow the concurrent execution of the agents. Therefore some basic means for code and data protection for the agent memory is required. This is achieved by assigning each of the agents an own memory segment and not allowing any other agent to access any other memory but its own. If some collaboration between the agents is required this must be requested via the communication service. The metadata of an agent as its code and memory segment will be stored in a structure that is shown in the figure below.

Every agent has a unique id within the virtualization platform. Additionally a priority and a status for the scheduler are stored. Assigning these values for an agent lies within the scope of an agent developer. Reproducing an agent on a virtualization platform where the agent's id

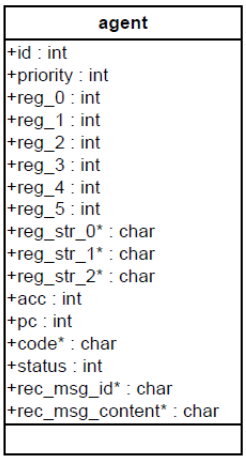

Figure 3.2: Agent structure

is already used will result in a denial of the reproduction by the platform. Every agent has 6 numerical general purpose registers used for the execution of the agent language. Additonally there are 3 char general purpose registers. The result of every agent language command will be written to the accumulator. There is also a program counter which is used for the execution of the agent and the numerical agent language representation is stored as well. The agent structure also contains a buffer for receiving messages from other agents and variable to store the id of the received message.
In order to reproduce the agent on another board or node the agent's structure needs to be serialized and transmitted via the communication layer.
Additionally the virtualization platform has to provide the agent developer with some means to deploy the agent executable to the virtualization platform, during compilation of the platform. During the initialization of the platform all deployed agents should be instantiated on the given platform.

3.3 Execution Layer

The execution layer is responsible for the execution of an agent which is written in the agent language and later translated to agent opcodes. The agent language provides means for:

- storing values to the general purpose registers
- comparing the contents of the general purpose registers
- performing basic mathematical functions like addition, subtraction, multiplication and division
- a jump operation
- reproduction and cloning functions
- sleep, delay and terminate functions
- functions to access the hardware attached to a node

If a function of the agent language returns a value, this value will be stored in the accumulator, where it can be used later on for further operations e.g. comparison etc.

The basic workflow of the execution layer as soon it is called by the scheduler is to read the next agent language opcode (all agent opcodes have a fixed length) as identified by the program counter, to decode it and to perform the function which is described by the opcode. Eventually the program counter value is changed and the control is returned back to the scheduler. The execution layer is called by the method execute which takes the following input parameters:

- Pointer to a specific agent structure
- Number of opcodes to execute

3.4 Hardware Services

The virtualization platform provides access to the hardware attached to a node via according hardware device drivers. The methods of the device drivers are made public to the execution layer which in turns allows the agents to access these methods. As the hardware supported by a node differs from node to node the virtualization platform should be able to discover during its initialization which hardware is supported on the node where it's running.
This will be achieved by defining a global set of function pointers within the virtualization platform. This set should contain all possible methods of all available device drivers. During initialization the platform will assign the according function pointer to a method provided by the device driver if the device is supported, otherwise the according function pointer will stay null.
All device drivers should support the method init_driver, which will be called by the platform during its initialization. Even if a hardware device is not supported on a given node, there should be at least a dummy device driver for it present, providing an empty init method. A real device driver should register its methods to the global function pointers during its init_driver method execution. Choosing this approach we would reach some form of modularity which would allow us to exchange to device drivers without necessity to change the platform code.

3.5 Communication Layer

The agents should be able to communicate with other agents on the same node or on the same board. Therefore the agent language provides means to request the sending or receiving of a message.
The sending function is blocking the further execution of the agent until the message is sent respectively received. When an agent wants to send a message this message is proceeded to the communications service which takes care of the actual transmission. While the sending procedure is ongoing the further execution of the sending agent is blocked. Its status in the agent structure is set to "blocked". As soon as the communication service signalizes a successful message transmission or a failure the result of the sending function is written to the accumulator and the agent will be made available for further execution.
When an agent sends a message to another agent, the receiving platform stores the id of the message and its content to the receiver agent structure. The receiving agent is able to retrieve the last message from its buffer. However only one message can be stored within the receiving agent structure and the next message will overwrite the content and possible the id of the last message.
The communication service provides no guarantees that sending of a message will succeed; it works on a best effort approach. Therefore the agent developer has to make sure by reading the return value of a sending operation whether the message was successfully sent or not and should initialize a retransmission in case of failure.
Every message sent should be identified by an id, in order to allow the transmission of messages with different semantics.

The receiver of a message should be identified via the node number (0..3) where the receiving agent is currently expected to be running and the receiver agent id. As the ids of agents are within the scope of the agent developer she has to make sure, that the correct receiving agent is addressed.Additionally a multicast message could be supported by allowing omitting the node address which should result in sending the message to all agents identified by the provided id.

3.6 Scheduler

The main task of the scheduler which is part of the virtualization platform is to identify the next agent to be executed and to utilize the execution layer to perform the execution of the according agent. The decision which agent to be chosen should be made on a static priority based scheduling policy.
Every agent is assigned a priority (0..3) by the agent developer which is stored within the agent structure. The highest priority is 3 and the lowest priority is 0. Based on the priorities of the currently running agents the scheduler creates a static list by which the order of the agent execution is defined. The scheduler instructs the execution layer to execute exactly priority + 1 opcodes for a given agent. Eventually the control returns to the scheduler and the next agent from the list is picked. The list is iterated cyclically.
If an agent which is to be scheduled next is currently blocked by a sending operation, then its execution should be omitted.
As an agent can reproduce it self to another node or board or clone itself within the same platform the scheduling list requires adaptation as soon as new agent is deployed on a platform. Whenever a given platform is the destination end point of a reproduction respectively cloning operation the scheduler needs to update its scheduling list before proceeding with further executions.

3.7 Agent language

3.7.1 Agent language (Assembler level)

To develop a mobile agent *Agent language* will be used. Language is tied to agent internal structure and support necessary operation for code mobility and message exchange. While writing the program for agent user should not be aware of hardware services presented on a given platform, but have common knowledge about all available services and what operations are allowed to do with the services (have list of services and available operations). It is the responsibility of the platform to provide required service to the agent (perform measurement, IO operation) or to manifest an error if the service is not available on the given platform. All current variables are allowed to store only in registers of agent structure (Fig. **??**).
The language supports the following groups of operations: arithmetic, control flow, code mobility, message exchange , access to hardware services.

Arithmetic operations of agent assembly language

Addition
Add the content of reg1 and reg2 (or value) and put the result into acc.

```
add reg1, reg2
add reg1, value
```

Subtraction
Subtract reg2 (or value) from reg1 and put the result into acc.

```
sub reg1, reg2
sub reg1, value
```

Division
Divide reg1 by reg2 (or value) and put the result into acc.
`div reg1, reg2`
`div reg1, value`

Multiplication Multiply reg1 and reg2 (or value) and put the result into acc.
`mul reg1, reg2`
`mul reg1, value`

Control flow operations and comparison in agent assembly language

Jump if greater
Jump to addr in code segment of agent structure if the value of acc. is 1.
`jmpgr addr`

Jump if equal
Jump to addr in code segment of agent structure if the value of acc. is 0.
`jmpeq addr`

Jump if less
Jump to addr in code segment of agent structure if the value of acc. is -1.
`jmpls addr`

Comparison
Compare reg1 and reg2 (or value).
If reg1 $>$ reg2 put 1 into acc,
if reg1 $=$ reg2 put 0 into acc,
if reg1 $<$ reg2 put -1 into acc.
`compare reg1, reg2`
`compare reg1, value`

Code mobility operations of agent assembly language

Move code
Move agent structure to platform that possess required service
`move service`

Clone code
Replicate agent structure on the given platform
`clone`

Die
Destroy agent structure and free corresponding memory
`die`

Message exchange

Send
Send acc. value to platform.
`sendmsg platform`

Receive
Pull message from platform to acc.
`pullmsg`

Store, move and wait operations

Store
Store value in register
Store character in register
`store reg, value`
`storecr reg, char`

Move
Move value from reg2 to reg1
`mv reg1, reg2`

Wait
Wait until event??
`wait`

Assign priority value
Assign priority of the agent to value in range 0..3
`priority value`

Access to hardware services

Set
Set service to reg or value
`setservice reg`
`setservice value`

Get
Put corresponding value from the service to the acc.
`getsetvice`

3.8 Communication Architecture

The communication architecture is designed to support communication between nodes on the same development board as well as between boards.

3.8.1 Hardware

The communication on the board is carried out over two serial bus channels. One of them is to be used for a distributed control application running on nodes 0-3. Another bus is dedicated for code mobility between nodes 0-4.

Access to the bus is controlled by separate UART modules on each micro-controller. The bit rate is constrained by the maximum value of 2 Mbps according to the manual.

Node 4 functions as a gateway to another board. It is a bridge between the local and the wireless zigbee network.

3.8.2 Distributed Control

Time-triggered protocols are customary for distributed control applications. This approach is suitable for low data volumes and data subject to real time constraints and regular sending intervals. In a time triggered scheme each node has a separate slot for writing to the bus. Meanwhile, other nodes can read the bus in the same slot or process a computation task.

3.8.3 Code Mobility

Code mobility between nodes includes local mobility on the same board and remote mobility between different boards. Executable agents generally have larger volume than control data. Sending at regular time intervals is not assumed, thus communication is aperiodic. A simple protocol based on message acknowledgment can be used.

There are two use cases: a) local mobility: destination is one of the nodes 0-3. b) remote mobility: destination is the gateway node 4. The gateway is to contain a zigbee stack implementation to enable access to the personal area network.

3.8.4 Addressing Scheme

Simple local addressing requires unique identifiers for each node. The requirement is that this be compatible with the time-triggered protocol implementation. For remote communication, board addresses have to be compatible with the configuration of the zigbee network. Since, each node will have a static number of agent execution environments, the address has to contain its identifier as as well.

3.8.5 Communication Interface

The interface for accessing the communication system is given below in Figures **??** through **??**.

```
typedef struct comm_msg_S {
  int type;          /* Message type identifier */
  int node;          /* Destination or origin node
                        depending on the context */
  int board;         /* Destination or origin board
                        depending on the context */
  long int len;      /* Payload length */
  char *payload;     /* Payload data*/
} comm_msg_T;
```

Figure 3.3: Message Structure

```
/**
    Function: comm_register
              Registers a communication endpoint
    Returns: Zero if successful
    Parameters: board
                Board id
                node
                Node id inside a board
 */
int comm_register(int board, int node);
```

Figure 3.4: Endpoint Registration

```
/**
   Function: comm_set_recvr
             Sets a callback to process incoming messages
   Parameters: recvr_callback
               Callback function that processes a received message
   Returns: Zero if successful
 */
int comm_set_recvr(void (*recvr_callback)(comm_msg_T *msg));
```

Figure 3.5: Message Receiving

```
/**
   Function: comm_send_msg
             Sends message over communication interface
   Parameters: msg
               Pointer to message structure;
               structure will be copied by function
               sent_cb
               Callback that is called when the message has been sent
   Returns: Zero if message is accepted for sending
            or nonzero if rejected

*/
int comm_send_msg(comm_msg_T *msg, void (*sent_cb)(int));
```

Figure 3.6: Message Sending

4 Implementation timetable

Figure ?? shows the implementation timetable.

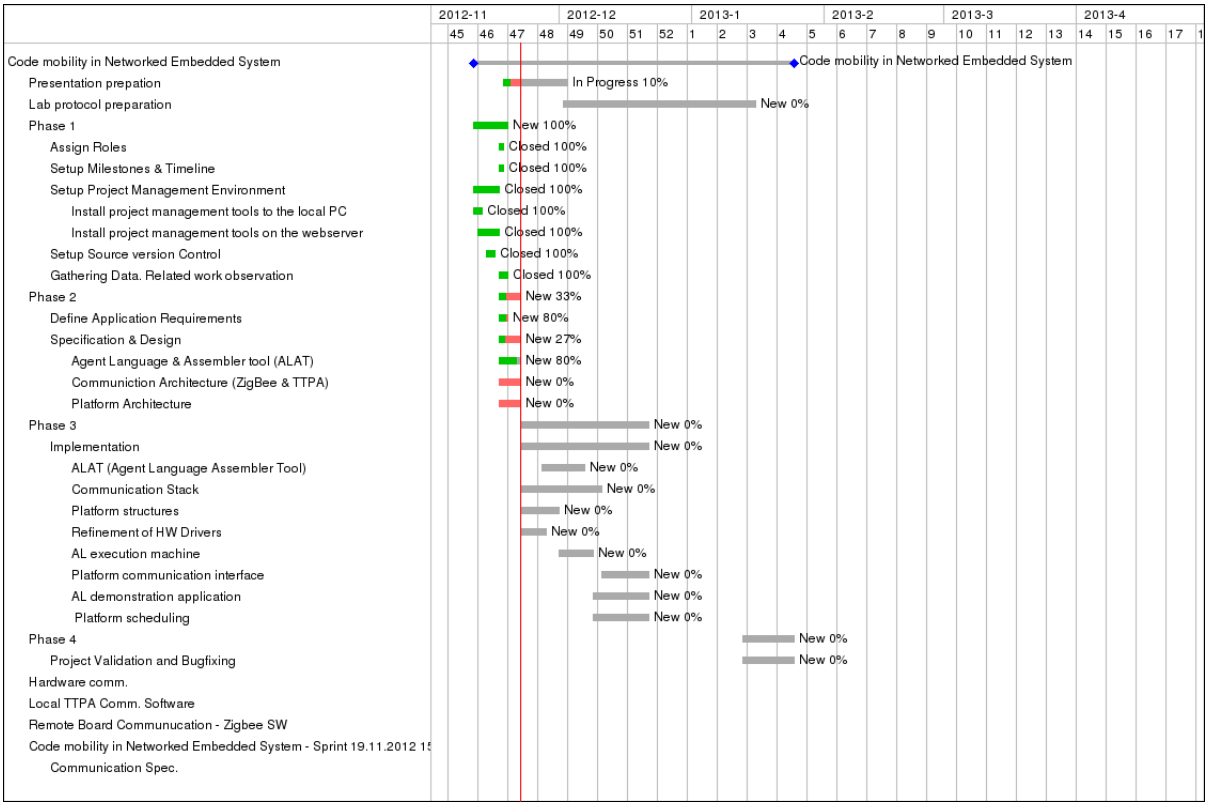

Figure 4.1: Gantt Diagram of the Project