Technical Report

urn:nbn:de:gbv:830–tubdok–12285

GlueAPI
Joining REFLEX and CometOS

Gerry Siegemund
Hamburg University of Technology
Stefan Lohs
Brandenburg University of Technology
Hamburg, Cottbus, Germany

2013

Hamburg University of Technology
Brandenburg University of Technology
TUHH

Table of Contents

Abstract

English Wireless sensor networks (WSN) are built for various tasks in arbitrary environments. Operating systems for sensor nodes appear in multiple forms, from a variety of vendors, universities, or private persons. This work presents the GlueAPI, a merge layer to combine the usability of two such operating systems: REFLEX of the Brandenburg University of Technology and CometOS of the Hamburg University of Technology. Both operating systems are suitable for simulations using the OMNeT++ framework and several hardware platforms.

Deutsch Drahtlose Sensornetzwerke (WSN) wurden für viele verschiedene Aufgaben in unterschiedlichsten Umgebungen entworfen. Betriebssysteme für Sensorknoten gibt es in diversen Formen, von unterschiedlichen Anbietern, Universitäten, oder Privatpersonen. Dieser Report präsentiert die GlueAPI, eine Zwischenschicht um zwei solche Betriebssysteme, REFLEX von der BTU Cottbus und CometOS von der TU Hamburg, zu verschmelzen. Beide Betriebssysteme unterstützen Simulationen mit dem OMNeT++ - Framework und einer Reihe von Hardwareplattformen.

Chapter 1

Introduction

We introduce the *GlueAPI*, an interface library that allows programming for the REFLEX [1] and the CometOS [2] operating systems for wireless sensor nodes (Figure 1.1). Both operating systems allow programming for different types of sensor nodes (i.e., hardware) and evaluating these algorithms in the OMNeT++ simulation framework. The programming language for the GlueAPI, the operating systems, and the simulation system is C++. Either CometOS or REFLEX and their dependencies are necessary for the GlueAPI to work. Knowledge of either system is not required.

The GlueAPI enables the user to send data between certain components of the system and through the network. It also provides message receiving and task scheduling. Memory allocation on the other hand is not modifiable and handled transparently by the underlying systems.

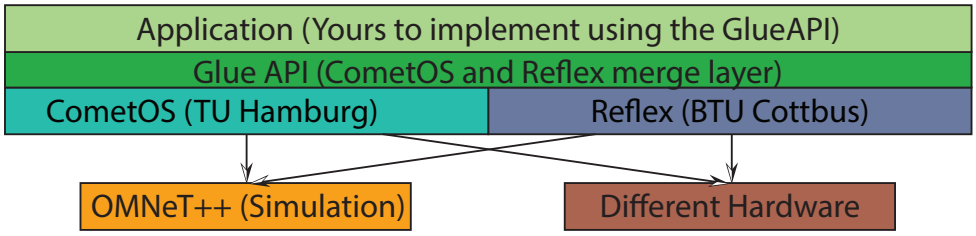

Figure 1.1: GlueAPI in architecture context

Chapter 2

Resources

GlueAPI The current version of the GlueAPI is part of the ToleranceZone repository. The URL of the repository trunk is `idun.informatik.tu-cottbus.de/toleranceZone/svn/implementation/glueAPI/trunk`. The files include the header files for both operating systems.

OMNeT++ and MiXiM For evaluating the system the OMNeT++ simulation environment is used. The GlueAPI was tested with the OMNeT++ 4.3 version (`www.omnetpp.org`). To simulate the behavior of a sensor network with wireless communication use the MiXiM framework in its current version 2.2.1 (`mixim.sourceforge.net`).

OMNeT++ is a simulation environment. Among other things, it is used to simulate networks, nodes, and the communication in and between parts of the nodes. Nodes are the main components of a network. A node consists of layers, or modules. The top module is usually referred to as application. All layers of a node are connected through gates. OMNeT++ uses ned-files to wire all components. Layers can have multiple gates.

REFLEX The REFLEX operating system is being developed by the distributed system/operating system group of the Brandenburg University of Technology (BTU). The current release and more information can be found at `idun.informatik.tu-cottbus.de/reflex`.

CometOS The CometOS operating system is being developed by the Institute of Telematics at the Hamburg University of Technology (TUHH). The current release and more information can be found at `www.ti5.tuhh.de/research/projects/cometos`.

Chapter 3

Interface Description

The GlueAPI consist of several header files which abstract the communication (`Gate`, `PlatformGate`, `Message`) and the scheduling (`Activity`, `TimeTriggeredActivity`) of the underlying operating system. The API uses the `namespace glueAPI`. In the following the interfaces of the GlueAPI classes will be described.

3.1 Module

Each C++-Class describes a component of the system. A component can contain different activities and communicates with other components by using message passing gates. To describe such a base component, e.g., the application layer, the `Module` interface needs to be implemented.

Methods

The `initialize()` method (Listing 3.1 line 3) initializes the component. After finishing the construction of all modules the `initialize()` method of all modules in the system is executed. The method is abstract and needs to be implemented.

```
1 class Module : public cSimpleModule {
2     virtual void initialize() = 0;
3 };
```

Listing 3.1: Module

Reflex

The interfaces provides a `#define` to define the module name. In case of REFLEX this pre-compiler directive does nothing.

CometOS

The interface provides the name to access the module by name at the `.ned` configuration files.

3.2 Message

To communicate between different components the `Message` datatype is used. This datatype represents a byte array of an static length `MESSAGE_SIZE`. The `buffer` field can not be written directly (because it is private). The `writePos` variable represents the size of the written buffer, it can be read with the `getLength()` function.

```
#define MESSAGE_SIZE 100

class Message {
   private:
      uint8_t writePos;
      uint8_t buffer[MESSAGE_SIZE];

   public:
      buffer[ PACKET_ID ];
      buffer[ APP_POS ];
      buffer[ STATE_POS ];
      buffer[ NB_POS ];
      buffer[ NODE_ID_POS ];

   Message();

   void setNodeID(nodeIdType ownID );
   void setPacketID( uint16_t val );
   void setAppPos( uint8_t pos );
   void setStatePos( uint8_t pos );
   void setNBPos( uint8_t pos );

```

```
23      NodeIdType getNodeID();
24      uint16_t getPacketID
25      uint8_t getAppPos();
26      uint8_t getStatePos();
27      uint8_t getNBPos();
28
29      uint8_t getLength();
30
31      void write( const Type &value );
32      uint8_t read( Type &value, uint8_t readPos );
33  }
```

Listing 3.2: Message

Using template functions, writing and reading of any type is possible, using the analogous functions. The `read` function returns `readPos + sizeof( Type )`, therefore consecutive reading, e.g., in a while loop is possible. The first six byte of any message are reserved for the header. It states where in the message which part of information is encapsulated (see Figure 3.1 for clarification). Not all possible message fields have to be used, e.g., it can consist merely of application data.

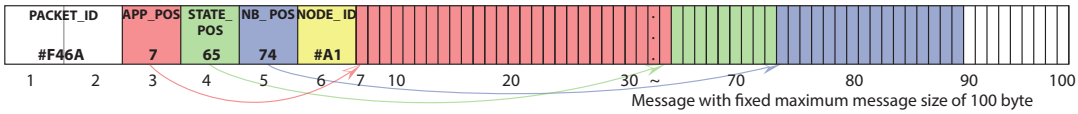

Figure 3.1: An example message, using all possible fields

Three different types of message data are supported. Application (APP) data, state data, and neighborhood (NB) data. With the obvious setter and getter methods a programmer can indicate where certain information starts and reproduce where it ends, respectively. The ending of this data is either the beginning of a subsequent field or the end of the message.

(Side note: The above is done to make piggy backing of, e.g., neighborhood data, easier. A programmer who wants to send application data only will not notice this extended functionality.)

3.3 Gate

The `Gate` class represents a message passing channel, i.e., the in/output of a component. Each component can contain an arbitrary number of gates, if necessary. The gate is a template

class, consisting of `ClassName` and a pointer to the handle (callback) function for messages receiving (Listing 3.3).

```
1 template<typename T, void(T::*MemFn)(Message*)>
2 class Gate {
3    Gate(T& object, const char *GateName)
4
5    void send(Message msg)
6 };
```

Listing 3.3: Gate

A `Gate` is declared by:

```
1 Gate<ClassName, &ClassName::handleReceiveFunct> gateName;
```

The `ClassName` is the name of the class of the owner object. This object must contain a member method, which handles the reception of a message from another component. An example for such a method is:

```
1 void handleReceiveFunct(Message* msg);
```

Methods

The Gate class needs to be constructed by the owner class (Listing 3.3 line 4). The constructor needs two parameters, the object which owns the gate and the name of the gate itself. The name is necessary to connect two gates and must be unique at the specific module.

With the send method (Listing 3.3 line 6) a message is passed from an out gate to the input of a target gate.

Beware!: Because of the typical lack of a memory management at an embedded system, the passed message is only a local variable. Because of the event driven behavior it is necessary to copy the message at the destination of the output.

Reflex

In case of REFLEX the system is not configured by `.ned` files. The connection of modules is done by a `NodeConfiguration`. To connect the gate output with the gate input of another gate two additional methods exists.

```
1  void connect_out_outGate(reflex::Sink1<Message>* ↙
       outGateDestination);
2  reflex::Sink1<Message>* get_in_inGate();
```

`connect_out_outGate` connects this gate output with the input of another gate. In case of REFLEX a gate abstracts from the sink concept. the `get_in_inGate()` methods returns a pointer to the input of the gate.

CometOS

CometOS behaves like a regular OMNeT++ simulation. The gates and the layers inside of a node are wired by `.ned` files. Additional configuration is done by the `omnetpp.ini`. There is nothing specific to be aware of when defining `gates` with CometOS because it is build right on top of OMNeT++.

3.4 PlatformGate

While the Gate is designed to pass messages/data between components of the application layers, the `PlatformGate` is designed to pass messages to the operating system. The usage of the `PlatformGate` is the sames as the `Gate` 3.3.

Reflex

The implementation of the `PlatformGate` converts a `glueAPI::Message` into a `reflex::Buffer` and vice versa. The buffer is passed to a `reflex::Sink1<Buffer*>` of the REFLEX part of the System.

CometOS

The behavior is the same as the normal `Gate` implementation.

3.5 Activity

This class provides scheduling as an activity function, e.g., if an event was triggered this activity could handle such an event. The definition of an `Activity` is shown in listing 3.4 line 4.

The `ClassName` is the class type of the owner object, the `handleMethod` is called if the system scheduler triggers the activity. The `handleMethod` member function must be of the form as shown at line 3.

The constructor 3.4 line 10 of the `Activity` class needs the owner object of the activity. The `notify()` method schedules the registered function for execution.

```
1  class app : public Module {
2     void handleMethod();
3     Activity<ClassName, &ClassName::handleMethod> activity;
4  }
5
6  template <typename T, void (T::* MemFn)()>
7  class Activity {
8     Activity(T& object);
9     void notify();
10 }
```

Listing 3.4: GlueAPI Activity class and usage

3.6 TimeTriggeredActivity

This class extends the `Activity` class by timer triggered scheduling. The definition and construction of the `TimeTriggeredActivity` is analog to the father class, `Activity`. In addition there are three new methods: 3.5.

```
1  template <typename T, void (T::* MemFn)()>
2  class TimeTriggeredActivity {
3     void setTime(uint16 time, timerType type);
4     void start();
5     void stop();                                          }
```

Listing 3.5: GlueAPI TimeTriggerdActivity class

The set methods initialize the `time` to the next scheduling of the activity. The repetitions are defined by the `timerType`. Possible values are `PERIODIC` or `ONESHOT`. The `start()`-method (re)starts and the `stop()`-method stops the timer. Note: A timer cannot be resumed, only restarted.

Chapter 4

Example

In the following a small Ping-Pong application, using the GlueAPI, is shown. It can be found in the GlueAPI folder mentioned in Section 2. The goal of this application is to send a Ping message to each all neighbors which reply with a Ping message of their own (forwarded to all their neighbors).

To demonstrate the usage of the GlueAPI the Ping message is send timer triggered. Listing 4.1 shows the header file of the application. It declares one `TimeTriggerdActivity`, two start and stop `Activity`s, and a `Gate` for communication. The `Gate` is connected to the `receivePacket` method for reception of packets.

```
1  #include "TimeTriggeredActivity.h"
2  #include "Activity.h"
3  #include "Message.h"
4  #include "Gate.h"
5  #include "Module.h"
6
7  class PingPong : public glueAPI::Module {
8  using namespace glueAPI;
9  public:
10    PingPong();
11    void initialize();
12
13    void handleTimer();
14    void receivePacket(glueAPI::Message* msg);
```

```
15
16     void start();
17     void stop();
18
19 private:
20     TimeTriggeredActivity<PingPong, &PingPong::handleTimer>
21         timerActivity;
22
23     Activity<PingPong, &PingPong::start> startActivity;
24     Activity<PingPong, &PingPong::stop> stopActivity;
25
26     Gate<PingPong, &PingPong::receivePacket> gate;
27 }
```

Listing 4.1: Header file of Ping Pong Application

The implementation of the constructor (listing 4.2) needs to initialize all the activities and gates. The activities are connected to the `this` object, to locate the appropriate function, as soon as an activity is triggered. The gate is connected to the `this` object and named "lowerInOut".

```
1 PingPong::PingPong():
2     timerActivity(*this)
3     , startActivity(*this)
4     , stopActivity(*this)
5     , gate(*this, "lowerInOut")
6 {
7     rounds = 0;
8     inactive = true;
9 }
```

Listing 4.2: Constructor of Ping Pong Application

After finishing all constructors the `initialize` (listing 4.3) method is run. If the current nodes id is zero, then the `startActivity` is triggered and the connected function `start()` is registered for scheduling. The `start` method sets the timer activity to a periodic timer with a duty cycle of one second (1000ms). The `stop` method stops the timer, if the stop activity is

triggered.

```
void PingPong::initialize() {
   // only node Id = 0 starts on its own
   // all other nodes have to receive a packet
   if (NodeID == 0 )
   {
      startActivity.notify();
   }
}

void PingPong::start() {
   // set time to 1s
   timerActivity.setTime(1000, timerActivity.PERIODIC);
   inactive = false;
}

void PingPong::stop() {
   // stop timer
   timerActivity.setTime(0, timerActivity.PERIODIC);
   inactive = true;
}
```

Listing 4.3: initialize of application module

After one duty cycle the `handleTimer` method is scheduled (listing 4.4). The method constructs a `Message` and writes the string “Ping” as application data then it is sent through the out gate.

```
void PingPong::handleTimer() {
   rounds++;

   glueAPI::nodeIdType nodeId = NodeID;

   glueAPI::Message msg;

   msg.setNodeID( platform->getNodeId() );
```

```
9
10      msg.setAppPos( msg->getLength() );
11
12      msg.write('P');
13      msg.write('i');
14      msg.write('n');
15      msg.write('g');
16
17      gate.send(msg); // send message
18
19      if (rounds > 10)
20      {
21          stopActivity.notify();
22      }
23  }
```

Listing 4.4: Timer handling of ping pong application

If the `Gate` receives a message it is passed to the defined receive method. Here the `handle` method starts the timer for ten rounds (listing 4.5). After each node, which receives at least one ping messages, sends ten ping messages the application stops.

```
1  void PingPong::receivePacket(glueAPI::Message* msg) {
2      if (( inactive ) && ( rounds < 10 ))
3      {
4          startActivity.notify();
5      }
6  }
```

Listing 4.5: Handle reception method

References

[1] K. Walther and J. Nolte. Xtc: A practical topology control algorithm for ad-hoc networks. In *Proceedings of the 21st International Conference on Advanced Information Networking and Applications Workshops (AINAW), Washington, DC, USA*, pages 784–791. IEEE Computer Society, 2007.

[2] Stefan Unterschütz, Andreas Weigel, and Volker Turau. Cross-Platform Protocol Development Based on OMNeT++. In *Proc. 5th Int. Workshop on OMNeT++*, March 2012.