COPRA - A COmmunication PRocessing Architecture for Wireless Sensor Networks

Reinhardt Karnapke, Joerg Nolte
{karnapke, jon}@informatik.tu-cottbus.de

BTU Cottbus

Abstract. Typical sensor nodes are composed of cheap hardware because they have to be affordable in great numbers. This means that memory and communication bandwidth are small, CPUs are slow and energy is limited. It also means that all unnecessary software components must be omitted. Thus it is necessary to use application specific communication protocols. As it is cumbersome to write these from scratch every time a configurable framework is needed. Copra provides such an architectural framework that allows the construction of application specific communication protocol stacks from prefabricated components.

1 Introduction

Sensor networks are collections of small sensor nodes with wireless neighbourhood broadcast facilities. Since sensor networks shall be deployed in large scales (possibly thousands of nodes [2, 1]), the overall cost dictates the use of cheap but simple radio transceivers for communication. The latter lack most of the common capabilities of WLAN or bluetooth networks. Even typical tasks like medium access control or the addressing of individual nodes in the direct radio neighbourhood are entirely left to software layers [5]. To make things worse the scarce CPU/memory resources of the sensor nodes do not allow to waste much space and processing power to process complex communication protocols [9]. Thus the designer of the communication software is stuck between a hard place and a rock: the simplicity of the radio requires much more work to be done by the CPU while the processing resources that are needed for this job are scarce. Consequently, communication protocols must be designed as close as possible to their intended use and the processing of the protocol stack must be dedicated to a specific user profile. However, designing application specific protocol stacks from scratch is always cumbersome and error prone.

This paper introduces Copra[1], an architectural framework for the construction of application specific communication protocols in wireless networks. In Copra often recurring protocol processing tasks are encapsulated in reusable components (so-called Protocol Processing Stages, PPSs) that can be composed to application specific protocol processing

[1] Copra is part of the Cocos Project which is supported by the German Research Foundation (DFG) in the SPP 1140.

engines (PPEs). Thus application specific protocols do not need to be designed from scratch but can be composed from prefabricated elements. The following sections are structured as follows: section two looks into Copra's structure, section three shows implementation details and section four briefly outlines other attempts in this area. We finish with a look at current status and future work in section five and the conclusion in section six.

2 The Copra framework

Copra is a library of protocol processing stages (PPSs) and a few already defined protocol processing engines (PPEs). A PPS is a special task in communication such as medium access, a PPE is a concatenated set of PPSs. By concatenating only the needed PPSs into a special PPE a lot of memory is saved. Each PPE can again be a part of a larger PPE. In figures one to three you will see examples of PPEs. The PPE seen on Figure 1 includes transceiving, medium access control and error checking, which normally is done by hardware. In case of sensor networks this part must also be managed by Copra because the cheap radios do not supply such functionality. The example on Figure 2 uses the broadcast PPE from Figure 1 as basis and adds address management. Note that the type of address is entirely configurable. It can be a number or a geographic location or even some property on the node, e.g. the value of the last temperature sampling. The third example on Figure 3 adds multi hop functionality to the PPE.

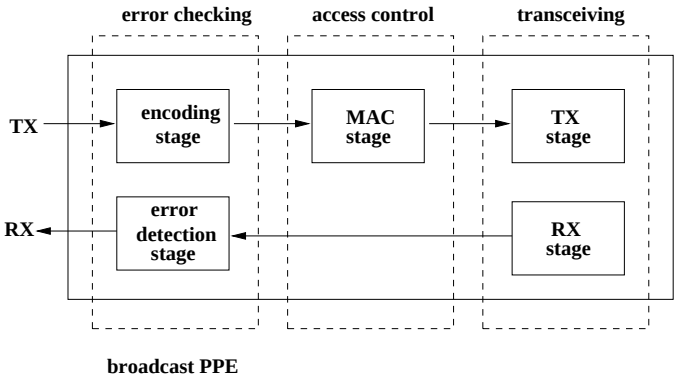

Fig. 1. A single hop broadcast PPE.

In the following sections we take a closer look at the two important parts of PPS and PPE.

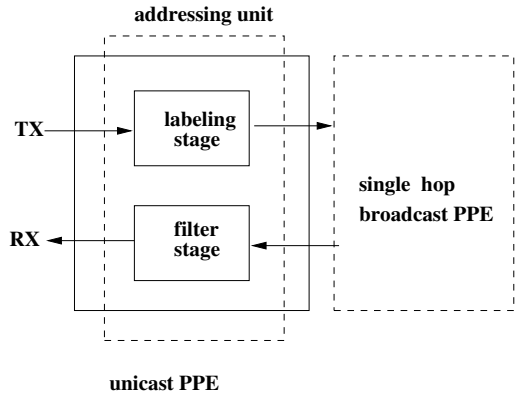

Fig. 2. A single hop unicast PPE

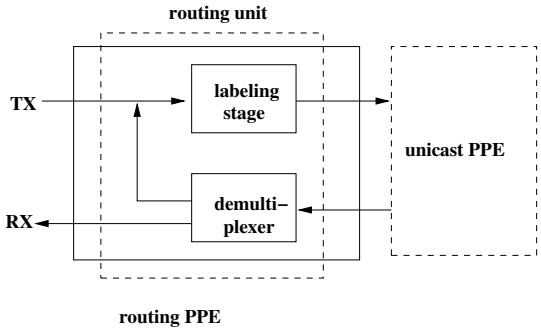

Fig. 3. A routing PPE

2.1 A protocol processing stage (PPS)

Most of COPRA's PPSs have a predecessor and a successor, with the exception of the end pieces of a PPE which have only one of them. PPSs normally consist of two parts which represent the direction the data flows: From the upper layers to the lower ones which is the transmitting (Tx) path and the opposing, receiving (Rx) path. To take this into account we provide the classes `RxStage` and `TxStage` from which a PPS has to be derived. An example for an end piece is the radio which does not have a successor because it transmits the data via hardware drivers. The data is represented as the data structure stack with the well known methods a stack supplies, the type of the stack is configurable as template parameter.

2.2 The protocol processing engine (PPE)

A protocol processing engine consists of a number of PPSs or other PPEs which are linked together. These links represent the transmitting and receiving chains which were already mentioned. Note that the layout is freely configurable. The end pieces of a PPE connect to the application on one side and the hardware drivers on the other. The `Radio` stage does not have a successor in the `TxChain` but uses the interfaces provided by the hardware drivers to transmit the data packets to another node. On the other node the `Radio` stage is the beginning of the `RxChain` and fills a stack with the data it receives from the hardware. The radio then forwards the stack along the `RxChain`.

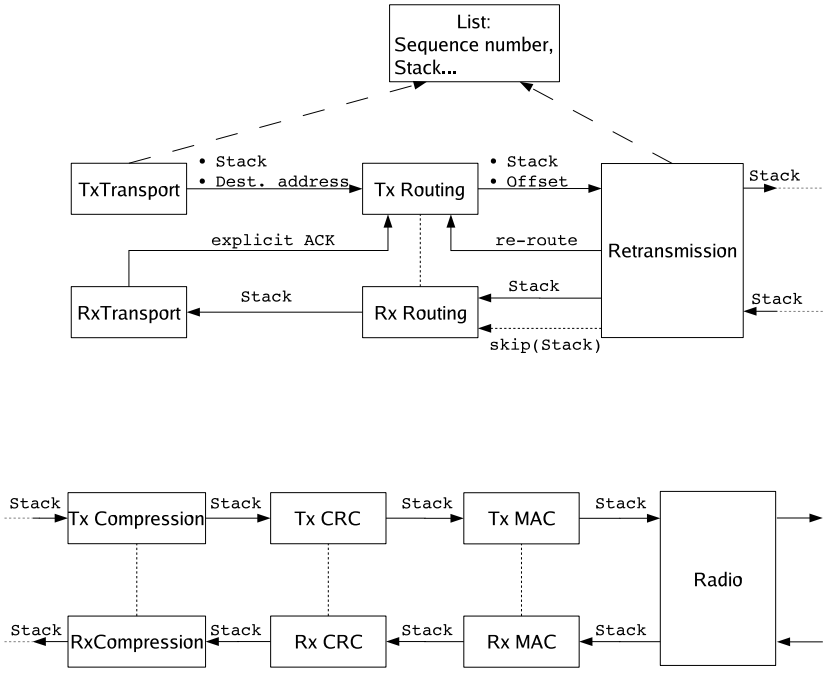

Fig. 4. A complex PPE which is used in our project

Figure 4 shows the largest PPE we have constructed yet. You might notice that the lower half of the picture which contains physical (radio), mac, error correction (crc) and compression follows the scheme mentioned before, where only `rxForward` and `txForward` are used. The upper half splits with the general concept as cross-layer issues arise. The `Retransmission` stage for example shares a data structure with the `Transport` stage. This is necessary because they use the same sequence numbers. The cross-layer issue between the `Retransmission` stage and

the `Routing` stage arises from the fact, that retransmissions may fail repeatedly. Then, the `Routing` stage is informed that it has to find new routes. Because of all these issues we tend to see the upper half of the picture as a single entity.

3 Applying the COPRA framework

When we want to use COPRA we configure it for a specific application. Lets assume that for this application we need to create a new PPE as none of the existing PPEs fits. Lets also assume that there is one particular PPS we need that does not exist either. For this reason we will now take a look at how a PPS is build.

3.1 Implementing a PPS

As example for a PPS the `FilterStage` is discussed here. Its job is only to forward incoming data packets on the `RxChain` if they are addressed to this node (including broadcast). Note that we are using reference counting to determine if the memory can be reused so we only decrement the reference count if the stack is unwanted. Please note also that the address is configurable as template parameter. This way it is up to the user to decide whether to use numbers, geographical identities ore even sensor values for addressing. As the `FilterStage` is a member of the `RxChain` it has to be derived from `RxStage`. In the `accept()` method the address of the destination is taken from the stack and compared to this node's id and the broadcast address. Only if one of these matches the stack is forwarded along the `RxChain`.

```
template<typename Address>
class FilterStage : public RxStage<Stack> {
 ...
 // called by previous stage in the RxChain
 virtual void accept(Stack* stack)
 {
  Address id;
  // get destination address
  stack->pop(id);

  // test if the packet is addressed to this node or the
      broadcast address
  if((id == myID) || (id == broadcastID))
   rxForward(stack);  // send stack to the next stage
  else
   stack->downRef(); // free memory
 }
}
```

In this example it is easy to see what a user has to do to construct a PPS. To build a member of the `Rx-/TxChain` the PPS has to be derived from `Rx-/TxStage`. The method in which all the work is done is called

`accept()` in the `RxChain` and `deliver()` in the `TxChain`. This is the only method the designer of the PPS has to fill. When all work is done the method `rx-/txForward()` has to be called, which delivers the stack to the next stage by calling `accept() (deliver())` on it. The forwarding methods are inherited so there is no need for the designer to touch these. They hide the identity of the succeeding stage.
Now that we have build the PPS lets take a look at how a PPE is constructed.

3.2 Composing a PPE

To build a PPE we need to have PPSs. As we have already build these we now have to connect them in the desired order. The following example is a datagram network (`DtgNet`). In this example you will notice that there are not a `RxRadio` and a `TxRadio` but only one `Radio` that works as both. The rxMac is omitted, because all a receiving MAC-layer would do is removing the MAC Header and we do not use any. This is because the data sampled by sensor nodes is normally small and we do not want to waste bandwidth and energy on unnecessary overhead. This PPE enables the application to use the standard way of sending by simply giving an address, a pointer and a length to the PPE's `send()` method. It also provides the method `receive()`, which allows the application to receive messages in the standard form. To receive a message the application supplies a buffer which should be filled with the message. After this is done, the number of received bytes is returned.
The connecting of the PPSs is done in the constructor of the PPE. First the receiving chain is built, then the corresponding transmitting chain follows. The methods `receive()` and `send()` are called by the application and offer the services mentioned above. They take care of memory management by selecting stacks from a pool and returning them once they are not needed anymore.
For simplicity reasons we omitted a few details, e.g. the check whether the buffer is big enough.

```
class DtgNet {

// the elements of the PPE
Pool pool; RcxRadio radio;
TxMac txMac; RxCRC rxCRC; TxCRC txCRC;
LabelingStage<Address> labeling;
FilterStage<Address> filtering;
MessageQueue msgQueue;

// Constructor.
// Here all parts of the PPE are assembled.
DtgNet()
{

 // build receiving chain
 radio.rxConnect(&rxCRC);
```

```
 rxCRC.rxConnect(&filtering);
 filtering.rxConnect(this);

 // build sending chain
 labeling.txConnect(&txCRC);
 txCRC.txConnect(&txMac);
 txMac.txConnect(&radio);
}

int receive(char* buf, int size)
{
 Stack* stack = msgQueue.get()
 if(!stack) // no message in the queue
  return 0;
 int used = stack->used(); // determine needed memory
 memcpy(buf, stack->tos(), used); // copy message
 stack->downRef(); // free memory
 return used; // return size of message
}

void send(char* msg, unsigned size, Address address)
{
 // try to get a new stack from pool
 Stack* stack = new (pool) Stack();
 if (stack) {
  void* buf = stack->alloc(size); // allocate memory
  memcpy(buf, msg, size); // copy message
  labeling.deliver(stack, address); // forward stack
 }
}
```

As you see it is very easy to construct a PPE. By calling `rx-/txConnect` on a PPS we connect it with its successor on the receiving (transmitting) chain. These methods are inherited from `Rx-/TxStage` so again there is no need to care for them. Also, in this example the great benefit of COPRA's modularity can be seen. Lets assume that the MAC Layer used above uses TDMA. Now we may need a different MAC for a different environment but all the rest should stay the same. We then replace the `txMac` with `txCSMAMac`. Now all we have to do is connect this stage instead of the original one and we are done. Another possibility to change this PPE would be to remove one unit, e.g. the addressing unit as seen in figure 2. All this is up to the user to configure. By supplying a variety of stages for all Layers we give the users an easy way to configure individual PPEs according to their needs.

3.3 Writing an Application

Now that we have PPSs and a PPE lets take a final look at the application. What the application does is of course up to the user but the easiest way to use a PPE will be discussed here. There are in fact two ways for

an application to use a PPE. One possible way is for the application to be the end piece of the receiving chain or the beginning of the transmitting chain. This way the application needs to inherit from `RxStage` or `TxStage` or both. This may seem a little drawback but it enables the application to use `txForward()` and work with the `accept()` method. It also has another advantage which will be seen when the second way is discussed. The second way is for the application to use a PPE with a special end piece, which allows the usage of standard communication interfaces. This end piece would offer a `send()` method which gets a pointer to the message and its size. In this method it would allocate a stack, copy the data and forward the stack. The advantage of this method is clear. The application does not need to worry about stacks, it does not even need to know it is using a PPE. The disadvantage lies in the end piece of the PPE. It has to copy the message to a stack which takes time. It also costs additional memory on the sensor nodes. An application would use the PPE seen above like this:

```
DtgNet net(myID);
Message msg;
...
// sending
net.send(4711, &msg, sizeof(msg));
...
// receiving
int size = net.receive(&msg, sizeof(msg));
...
```

Please note again that while in this example the address is a number it is entirely up to the user what type of address is being used.
Now that we have seen how the Copra framework can be used, lets take a look at the cost of using it.

3.4 Code Size

As mentioned above sensor nodes are limited in memory and have slow CPUs. In this section we take a closer look at the size of our framework. There are two figures which go into the code size. First, the size of the code which is independent of Copra as it would exist even if the framework was not used. Second, the overhead of using the framework. This overhead can be determined as follows:
Each stage has a pointer to its successor, the connecting method and the forwarding one. Also a `vtable` is needed for the inherited functions and the calls to the connecting methods must be made. Finally the call to the constructor of the PPE in which the connections are made needs to be considered.
Two things are included for every PPS, the pointer to the next stage and the `vtable`. The size of these depends upon the CPU in use. In our experiments we use Lego RCX robots [6] which include a Renesas H8/300 processor. This is a 16 Bit processor with a clock frequency of 16 MHz. On a 16 Bit processor the size of a pointer is two bytes which

means that the overhead for one PPS includes 2 bytes for the pointer to the next stage, 2 bytes for the pointer to the `vtable` and 6 bytes for the `vtable` itself. Altogether this means an overhead of 10 bytes per PPS. There are also the inlined connecting and forwarding methods and the constructor of the PPE. As these exist only once for the framework they are not taken into account here.
The next figure shows code sizes of two selected PPEs. The sizes were measured on the RCX robots we used for our experiments. As these sizes are dependent on the CPU in use they may vary on different systems.

PPE	buffer pool	radio	mac	crc	addressing	size (bytes)
broadcast	x	x	x	x		3400
unicast	x	x	x	x	x	3848

4 Related Work

In sensor networks the communication cost is reduced by replacing part of the communication with local computation. While this is a great improvement in battery lifetime it also means that the communication must be done in an application specific way. The authors of [4] call for a family of protocols for general purpose sensor nets. With Copra such a family exists, as the framework represents a lot of different communication protocol stacks that can be configured according to the applications needs. Copra is partly inspired by CORBA and .NET. The channel sink chains in .NET are configurable, meaning that the user can insert whatever sink he needs. These chains are reflected in Copra's `Rx-/TxChains`. An important difference is however, that Copra's chains starts where .NETs sinks end. The lowest of .NETs sinks is the `TransportSink`, whereas Copra is a communication framework. The portable interceptors in CORBA were also an inspiration, as it is possible to insert additional interceptors into a chain. This is reflected in Copra's PPSs which are connected in a PPE. While CORBA has a predefined order, the PPSs in Copra can be inserted anywhere in a PPE.
In [3, 7] the lack of an overall sensor network architecture is remarked. The authors describe the need for a sensor network protocol which should be located lower than the IP-Layer in the internet. While this so called SP should provide a set of functionalitys it should still stay configurable and be open to cross-layer issues. Copra offers the configurability and openness required.

5 Current Status and Future work

At the moment we have 14 different PPSs and 8 PPEs. While this number may not seem very large, it is not necessary for it to become much larger. We are experimenting with some of our PPSs and PPEs on modified RCX robots. These Robots have been additionally equipped with an easy radio ER400TRS radio module which we use instead of the included infrared

module (IR). To enable this, a serial port has been inserted which allows us to connect either the IR or the radio module. The IR is still needed to boot the RCX robots but once they are booted we switch to the radios. COPRA is independent of the operation system used, but we decided to use our self developed miniature OS `Reflex`[10] as basis. `Reflex` supports pre-emptive scheduling and provides hardware drivers which we use in some of or PPSs.
In the near future we will need to implement a few more different PPSs for each layer. Once we have these there could be more PPEs and application examples. But it is not our focus to find new applications for sensor networks, only to offer an easier way to build them. Also it is not our goal to build lots of PPEs. That is not necessary as the users will build their own ones. Right now we are using the RCXs only but we are going to equip these with ScatterWeb[8] sensor nodes. This is necessary because the RCXs have only three input channels and the additional serial port, which are connected to touch sensors and the radio. When we connect the ScatterWeb sensor nodes with the serial port we will be able to use their radio and have their additional sensors.

6 Conclusion

COPRA is an easy to use framework which allows a user to plug and run communication protocols for sensor nodes without having to rewrite the application each time a different hardware is used or the environment is different. Developers can now focus their attention entirely on the application. COPRA performs well in our experimentation environment and we are positive that it will work equally well in the next experiments using the ScatterWeb sensor nodes.

References

1. Ioannis Chatzigiannakis, Sotiris Nikoletseas, and Paul Spirakis. Smart dust protocols for local detection and propagation. In *POMC '02: Proceedings of the second ACM international workshop on Principles of mobile computing*, pages 9–16, New York, NY, USA, 2002. ACM Press.
2. Ioannis Chatzigiannakis, Sotiris Nikoletseas, and Paul G. Spirakis. Efficient and robust protocols for local detection and propagation in smart dust networks. *Mob. Netw. Appl.*, 10(1-2):133–149, 2005.
3. David Culler, Prabal Dutta, Cheng Tien Ee, Rodrigo Fonseca, Jonathan Hui, Philip Levis, Joseph Polastre, Scott Shenker, Ion Stoica, Gillman Tolle, and Jerry Zhao. Towards a sensor network architecture: Lowering the waistline.
4. John Heidemann, Fabio Silva, and Deborah Estrin. Matching data dissemination algorithms to application requirements. In *SenSys '03: Proceedings of the 1st international conference on Embedded networked sensor systems*, pages 218–229, New York, NY, USA, 2003. ACM Press.

5. J. M. Kahn, R. H. Katz, and K. S. J. Pister. Next century challenges: mobile networking for s̈mart dust:̈ In *MobiCom '99: Proceedings of the 5th annual ACM/IEEE international conference on Mobile computing and networking*, pages 271–278, New York, NY, USA, 1999. ACM Press.
6. Holly Patterson-McNeill and Carol L. Binkerd. Resources for using lego mindstorms. In *Proceedings of the seventh annual consortium for computing in small colleges central plains conference on The journal of computing in small colleges*, pages 48–55, , USA, 2001. Consortium for Computing Sciences in Colleges.
7. Joseph Polastre, Jonathan Hui, Philip Levis, Jerry Zhao, David Culler, Scott Shenker, and Ion Stoica. A unifying link abstraction for wireless sensor networks. In *SenSys '05: Proceedings of the 3rd international conference on Embedded networked sensor systems*, pages 76–89, New York, NY, USA, 2005. ACM Press.
8. Jochen Schiller, Achim Liers, Hartmut Ritter, Rolf Winter, and Thiemo Voigt. Scatterweb - low power sensor nodes and energy aware routing. In *Proceedings of the 38th Hawaii International Conference on System Sciences*, 2005.
9. K. Sohrabi, V. Ailawadhi, J. Gao, and G. Pottie. Protocols for Self Organization of a Wireless Sensor Network. *IEEE Personal Communication Magazine*, 7:16–27, October 2000.
10. Karsten Walther, Reinhard Hemmerling, and Jörg Nolte. Generic trigger variables and event flow wrappers in reflex. In *ECOOP - Workshop on Programming Languages and Operating Systems*, Jun 2004.