October 1, 2008

Notes To Master Thesis "The Progress Run-time Architecture"

Contents

1 Introduction **2**
- 1.1 The PROGRESS and the ProCom Model 2
- 1.2 Code Synthesis 2
- 1.3 Thesis Purpose 2

2 Channels **2**
- 2.1 Overview 2
- 2.2 Exact Semantics of Channel Writing and Reading 3
  - 2.2.1 Writing 3
  - 2.2.2 Reading 4
  - 2.2.3 How Could Be Another Semantics Achieved 4
  - 2.2.4 Time Synchronization 4
  - 2.2.5 Time Analisis of a Writing and Reading 4
- 2.3 Efficiency 5
- 2.4 General Design of Transfer Mechanism 5
- 2.5 Design of Transfer Mechanism in Demo Implementation 6
- 2.6 API to IDE User Programmer 6
  - 2.6.1 Similar Stuff for C programmers 6
  - 2.6.2 Socket like API Example 6
- 2.7 Channels implementation 6

3 Virtual Node Environment **6**
- 3.1 Passing Arguments to VN Code 6
  - 3.1.1 Example - Writing To the Channel 7
- 3.2 Inicializing the Environment 7
- 3.3 Accessing Hardware - Hardware Abstraction 7
  - 3.3.1 Example: Persistent File 7
- 3.4 Accessing Hardware - Synchronization 8
- 3.5 Accessing Channels 8
- 3.6 Virtual Node Code Structure 8

4 Bulding Physical Node Executable 9
4.1 Main Routine Structure . . . 9
4.2 Mapping one VN to Physical More Then One Time . . . 9
4.3 Generated Code . . . 9
4.4 Precompiled Code . . . 10

5 System Start Check 10

6 Opened Questions 10
6.1 Passing Data To Virtual Node . . . 10
6.2 Initializing Hardware (and Runtime Access to HW Resources) . . . 10
6.3 Accessing Resources - Resource Abstraction . . . 10
6.4 Accessing Channels . . . 11
6.5 Virtual Node Internal Structure . . . 11
6.6 Building Code for Physical Node . . . 11
6.7 System Start-up . . . 11

7 Sample Application 12
7.1 Hardware . . . 12
7.2 Operating System . . . 12
7.3 Application description . . . 12
7.4 Requirements to application . . . 14
7.5 Channels in application . . . 14
7.5.1 Design of the transfer mechanism . . . 14
7.5.2 Local Data Transfer Implementation . . . 16
7.5.3 Ethernet Data Transfer Implementation . . . 16
7.5.4 Channel Testing Application . . . 16
7.6 Code of the VN . . . 16
7.7 Notes . . . 18

1 Introduction

1.1 The PROGRESS and the ProCom Model

ProCom model will be shortly described here.

Models and components have become an indispensable part in the development of embedded systems. They reduce the complexity of embedded systems and provide a formal ground on which analysis and synthesis may be performed. This thesis is part of a large project, called PROGRESS, which aims at providing component-based techniques for the development of embedded systems.

1.2 Code Synthesis

The design and development of components in PROGRESS is supplemented by deployment activities consisting of four parts: (1) allocation of components to virtual nodes, (2) synthesis of the code belonging to each virtual node, (3) mapping virtual nodes to physical nodes and (4) creating of the glue and system wrapper code for previously synthesized virtual node code. In code synthesis, the code of components are merged, optimized and mapped to artifacts of an underlying real-time operating system. In creating of glue and system wrapper code, the communication between tasks from different virtual nodes is solved and main executable file responsible for running appropriate virtual nodes for each physical node is created.

1.3 Thesis Purpose

The aim of the thesis is to identify necessary questions about patterns for API and runtime environment of the synthesized code created in the second phase of the development process described above.

The thesis should investigate the structure of virtual nodes and the supporting mechanisms needed to run them on the physical nodes, including the support for communication between virtual nodes. A part of the thesis will be an implementation of a simple application based on the component approach which will help in identifying concerns and demonstrate possible solutions. The implementation should cover the handling of local and remote communication, event driven and timer driven tasks and running system with two physical nodes of embedded hardware.

2 Channels

2.1 Overview

Channels are only way how can the subsystems can comunicate one to each other. We can divide channels by different criteria. Chanel can be

Distributed Writter and reader are executed on different physical nodes

Local Writter and reader are executed on same physical node

Another point of view is the number of readers and writers of channels associated with the channel.

1–1 Simplest form, where one reader writes data, and one reader is reading it.

1–n One writer and many readers.

m–n Many writers and many readers.

From the developers (users of ProCom IDE) point of view the channel is one entity, something like one typed piece of memory, where any subsystem connected (for writting) can write and any subsystem connected (for reading) can read from. The channel implementation should not ensure any hard time synchronization.

2.2 Exact Semantics of Channel Writing and Reading

The exact semantics of channel operation can be different for different application. In the next paragraphs we will discus how could this semantic look like, and how could be achieved. We will also discus behavior usefull in common applications.

2.2.1 Writing

Data writing and triggering is indivisible operation. Any time, data are written to channel, triggering action is also performed. If application need to split these two actions, it's responsibility of this particular application to create a buffer and wait for triggering to write this buffer into the channel.

For different type of application we need different behavior of the channel. In some application we need to synchronize producer and consumer, we will need blocking semantic of this actions. In other application the task couldn't be blocked, so write will be non blocking and simply will overide existing data.

There are at least these semantics for channel writing, which could be usefull:

Blocking writing , the writer will be blocked, until the reader consume previous message.

Nonblocking writing , the writer will be never blocked. If the consumer not consumes data, the new data will be dropped or will overide previous data.

Reliable writing , where some mechanism exists, which ensure writer, that reader read data.

Unreliable writing , where writer don't care if anybody read or not.

For many of this flavors we can consider buffered and not buffered approach. For example channel, with internal capacity of 5 messages, read in FIFO style, where we rewite blocks when all buffers are full.

2.2.2 Reading

Situation with reading is very similar to writing. We can (at least) consider:

Triggered reading Triggered readers will read data any time, when triggering action is performed. This type of reading matches the event-driven task type. The data can not be lost, the channel implementation take care, that all the data written to the channel will be delivered to the reader. Triggered reading is a blocking operation.

Non-triggered (Random) reading Non triggered readers can read the current value of the data any time, they want. This type of reading matches the timer-driven task type. The channel implementation ensures, that the data will not be corrupted (a.g. part rewritten by next writing), but nothing else is guaranteed.

Unique and non-unique reading In the unique reading all the messages could be read by the reader only one time. This is direct analogy to reading from a socket. One time you read the data, you can not read it once more. Non unique reading will return data repeatedly. Every time the most current one.

2.2.3 How Could Be Another Semantics Achieved

Because the semantics of reading and writting can differ, there must be some mechanism how to achieve this different behavior introduced. Because channels are in many signs very similar to UNIX file descriptors, we can inspirate ourselves, how is problem solved there.

By API One possible way is to have another calls for each semantics of the operation.

By Attributes Attributes could be assigned to channels and connectors. This attributes can change operation behavior.

Hybrid Hybrid approach mix API and Attribute approach.

We will focus to to this topic in the section about channel design.

2.2.4 Time Synchronization

The channel implementation should does not ensure any hard time synchronization. This is not a goal of ProCom model. If there is a need for some kind of hard time synchronization, it have to be be realized by another mechanism, not by a channels.

2.2.5 Time Analisis of a Writing and Reading

Channels are just a way how could executable code of one virtual nodes comunicate one to each other. When we create, synthesize and analyze virtual node, we want all the tasks from this virtual node to have same behavior independently on the topology of the channels, where are they writing.

This mean, that there has to be some upper estimation for time needed for read() and write() operations, and this estimation can not be dependent on the topology of the channel (a.g. type of transfer mechanism in the channel or number of readers or writers).

One possible approach, how could be such behavior reached, is to have (at least) one task in the system, which will take care about delivering of the messages. This task (we will call it delivery task) can be blocked all the time when no messages are comming, and write operation just write the data to input buffer of the channel and wake up this delivery task.

Logical consequence of this approach is, that even if we map only one virtual node to physical node, the utilization of the system must be $U < 1$, because there have to reamain capacity for this delivery task.

Another way how to archive the estimation mentioned above is just to say, that the application will be only using some topology of the channels and made upper estimation for this set of channels.

This idea is very important for time analysis of the system, because if the call of write or read take constant time, each virtual node can be analyzed only once. The analysis of whole system then will be much easier, because the analysis of the virtual nodes will be already done and we will only analyze the behavior of tasks specific for this particular system.

2.3 Efficiency

Channel design patterns should efficient as possible, when it is possible should use efficient methods of underplaid OS.

2.4 General Design of Transfer Mechanism

From the logical point of view, there exist in the system two entities. The channels and the connectors. It is reasonable to use this logical separation also in the runtime design.

Reasons for this separation are at least this

1. Connectors can have attributes as well as channels.
2. Writing to connectors and not to channels can show up many mistakes in the compile time (a. g. writing to input connector).

Every input and output connector will be associated with ID. This ID will be internal for each subsystem. System wrapper code will take care of mapping this internal numbers to some system wide ID and associating connectors with channels. For example, inside a virtual node, you will know, that you are writing to output connector number two, and you don't care if there are more readers, more writer, if the channel is local or distributed. This decision is made because of reusability of components. This output number is internal for the virtual node, where is used. Un another virtual node can be also output two available, but this output connector can write to another channel.

2.5 Design of Transfer Mechanism in Demo Implementation

Is described in detail in section "application"

2.6 API to IDE User Programmer

2.6.1 Similar Stuff for C programmers

C programmers are inhabited to filedescriptors - like reading and writing and for message sending via mail boxes (typical in realtime systems), so the API should be little bit similar to one of this.

2.6.2 Socket like API Example

```
/* run triggering on channel */
int connector_write_triggered(SYS_OUTPUT_CONNECTOR connector, const void *msg);

/* reading from the channel, block until data ready */
int connector_read_triggered(SYS_INPUT_CONNECTOR connector, void *buffer);

/* reading from the channel, not blocking, does'nt ensure anything */
int connector_read_nontriggered(SYS_INPUT_CONNECTOR connector, void *buffer);

/* checks, if the data are ready, non blocking */
int connector_block_until_data_ready(SYS_OUTPUT_CONNECTOR connector);


/* checks, if the data are ready, non blocking */
int connector_is_data_ready(SYS_OUTPUT_CONNECTOR connector);

/* checks, if the data are ready, blocking. Blocks next execution until data ready
int connector_block_until_data_ready(SYS_OUTPUT_CONNECTOR connector);
```

This API can change a bit.

2.7 Channels implementation

Local implementation is **hardly!!** dependent on underlying OS. More about my implementation about implementation on Linux OS you can read in section about sample application.

3 Virtual Node Environment

3.1 Passing Arguments to VN Code

At least reasonable argument could be reference to inputs and outputs of VN. There is a question, if any other arguments should be needed at all in component VN code..

We can not be sure, that every system allows explicitly to pass some data into task. But many of them do(a. g. POSIX).

In the other cases we can use global variables (for example defined by some macro).

3.1.1 Example - Writing To the Channel

This fragment of the code, shows how can be readability of the code achieved by using macros for passing information about the channels.

```
connector_write_triggered(WRITER3_OUTPUT_0, (void *) &c);
```

WRITER3_OUTPUT_0 is macro defining reference to appropriate channel. This macros are defined in system generated code.

3.2 Inicializing the Environment

Initialization of the environment of the VN has two principial parts:

Software initialization , where variables are set to initial values, memory is allocated...

Hardware initialization , where initial values are written to the registers on hardware.

Software initialization could be done independently on the other virtual nodes.

Hardware initialization is problematic. What to do, when one node want to write 0 to some register as initial value and other 1? Or simply deny access hardware in in initialization? Or each Hardware resource can be accessed only by one VN? **Question above**

3.3 Accessing Hardware - Hardware Abstraction

Not every virtual node could run on every physical node. The code of the virtual node (or at least components from which was synthesized) should not be dependent on the physical hardware, to enable reusability of the components.

Now I'm looking on this problem like this: Each component is written for some virtual node. This VN has a list of hardware requirements. Every Requirement enable developer to use set of routines. Every type of physical node, where VN mentioned above could be mapped will have some kind of resource access library, which will implement this sections.

But this will be probably for longer discussion

3.3.1 Example: Persistent File

Component will be written for VN, with hardware requirement "Persistent storingfiles". This will enable for developer functions "file-write()", "file-read()". This functions would be implemented in library progress-nv-map-somepharchitecture.la, which would be specific for the physical node.

3.4 Accessing Hardware - Synchronization

How to ensure that there will be no collisions on the hardware access? Each Hardware resource could be accessed only by one VN? Or some kind of synchronization? **Question above**

3.5 Accessing Channels

Viz Channels.

3.6 Virtual Node Code Structure

This is an example, how the code of one task o some virtual node could look like (at least in my sample application).

Code of the virtual node is composed of tasks. This tasks are defined by one C function of given type. The type of the function allows to pass any data inside the task and also return pointer to any data.

The wrapper function task_run() will return in time, when all the tasks from this virtual node finish its work.

```
/* one task of virtual node */
void* vnode_writer_task_1(void * unused){
    char c;
    c = sensor_reader();
    while (c != ' '){
        printf("WRITER: writing to OUTPUT_0 (channel %d) '%c'\n",(WRITER_OUTPUT_0.
        fflush(stdout);
        connector_write_triggered(WRITER_OUTPUT_0, (void *) &c);
        c = sensor_reader();
    }
    connector_write_triggered(WRITER_OUTPUT_0, (void *) &c);
    printf("WRITER: writing to OUTPUT_0 (channel %d) '%c'\n",(WRITER_OUTPUT_0.chan
    fflush(stdout);
    return NULL;
}

#define WRITER_NR_TASKS 1

/* routine, which will run virtual node. */
void * vnode_writer_run(void * unused){
    int i;
    task_handle_t tasks[WRITER_NR_TASKS];
    tasks[0] = task_run( vnode_writer_task_1 );
    for (i = 0; i < WRITER_NR_TASKS; i++){
            task_wait(tasks[i]);
    }
    return NULL;
```

```
}
```

4 Bulding Physical Node Executable

4.1 Main Routine Structure

The main routine for each node runs all the virtual nodes. There is no reason, why should main function know internal structure of the synthesized code of the virtual node. It's only responsibility is initialize hardware, and run virtual nodes.

Example how the function look like in sample application.

```
#include <stdio.h>
#include "phnode2.h"
#include "vnode.h"

#define NR_TASKS 3

int main () {

    vnode_handle_t vnode_handles[NR_TASKS];
    int i;
    phnode2_init();
    vnode_handles[0] = vnode_run( vnode_reader3_run );
    vnode_handles[1] = vnode_run( vnode_writer2_run );
    vnode_handles[2] = vnode_run( vnode_writer3_run );
    printf ("Start process OK..\n"); fflush(stdout);

    for (i = 0; i < NR_TASKS; i++){
        vnode_wait(vnode_handles[i]);
    }

    phnode2_destroy();

    return 0;
}
```

4.2 Mapping one VN to Physical More Then One Time

Just note: run same task with parameters. It's not possible to link the same code two times to same binary.

4.3 Generated Code

Some code for building the binary for physical node could be precompiled, but still there will be several files, which will be generated. In this files is mostly channel

stuff done (so this files represent system connection schema - how are components connected one to each other) and mapping VN to physical nodes is done.

4.4 Precompiled Code

All other code could be precompiled (for example connectors, channels, mutexes, tasks, vnode management...).

5 System Start Check

Will be some extra checking performed after start-up of all physical nodes?

Is possible to controll all the channels? (Writer don't know who is listening, but channel entity knows). **Question above**

6 Opened Questions

6.1 Passing Data To Virtual Node

- It is necessary to pass at least information about input and output connectors of the channels.
- Is there a reason to pass any other data? Or VN is "single purpose" unit and doesn't need parameterization?
- Is there direct support for passing data to task in all system we want to support?
- Pros and cons of global variables and task parameters - Isn't using global variables for passing (hidden in some macro) more confortable then using parameters?

6.2 Initializing Hardware (and Runtime Access to HW Resources)

- Can more then one VN access same HW resource at runtime?
- If so, who will be responsible for synchronization?
- If not, how the mapping VN to physical node will solve collisions?
- Hardware initialization. Who is responsible for initialization of system HW environment? Collisions in initialization of shared resources.

6.3 Accessing Resources - Resource Abstraction

- Level of abstraction? (signal_ok() vs. led_access(GREEN_LED, SET_ON) vs. gpio_reg_set(GPIO_LED_GREEN))
- How to formalize requirements?

- Where will be the information about requirements stored? How can be accessed? How will be appropriate functions (and constants and types ...) enabled?
- Where the abstraction will be realized? In library for the hardware and library for the platform?

6.4 Accessing Channels

- Exact semantic of the channel operation..
- How could be this semantic achieved? By API? By attributes of channel or connector? Where to store this attributes?
- Channel API. Should it be similar to well known API's? (sockets? message_boxes? ...)
- Time Analysis Issues. How to achieve constant time to write and read to channel if don't know number of readers and writers? (This will be probably needed for time analysis) Extra "delivery" tasks? Or restrict channel topology to some types and make upper estimation?

6.5 Virtual Node Internal Structure

- How will the internal structure look like? Just a set of task? All runs in one VN wrapper?
- How the end of the execution of the VN will be signaled? (to release resources correctly)
- Should we be able to access (or stop or restart) some single task?
- Reentrance of the libraries code issues.

6.6 Building Code for Physical Node

- Build single executable for each node?
- Which code needs to be generated each time? Which could be precompiled? How to build executable for more platforms?
- How to initialize HW (which parts)?
- Should we be able to access (or stop or restart) some VN? Or some single task on this level?
- How the end of the execution of all the VN will be signaled? (to correctly release resources)

6.7 System Start-up

- Will the system perform any special handshakes betwen physical nodes? Or only channels control connections? Or nothing?

7 Sample Application

7.1 Hardware

Sample application will finally run on two embedded Arcom VIPER boards. These boards have four swithces and three leds on. This parts will simualte sensors and and outputs.

Inputs and outputs are realized by four switches and three led diodes on the board.

7.2 Operating System

On the boards is running Arcom Embedded Linux Operating system. This is not a realtime system, but API used for creating tasks (posix threads) is same as in many RT operating systems.

Inputs and outputs are realized by four switches and three led diodes on the board.

7.3 Application description

On the Figure 1. is drawn sample application.This application is composed of three (four) primitive ProSys components, two of them are connected into one composite component. This component model subsystems from the developers point of view

The basic functionality of the application will be very simple. Subsystems were synthetised from one or more ProSys components during previous code synthesis.

Invoker This subsystem will have two tasks. First will read data from sensor. Every time data from sensor came, this subsystem will write data read into channel connected to Responder subsystem. So this task will be event driven.

Second invoker task will be timer driven. Periodically will read the answer from responder, and if this answer will from the previous one, it will be forwarded to displayer.

Displayer subsystem will display results of the Invoker subsystems, concretely it will switch on and switch off led diodes on the board.

Displayer 2 This subsystem may be will not be implemented, it's here just to think about this case. It will be a copy of of Displayer component, but on another virtual node. It is here to demonstrate functionality of chanel with more than one reader.

Responder subsystem will receive the data from Invoker component and will generate the answer. Generate answer means perform some trivial operation with it, for example, if input will be 'c' character, return '!' character, otherwise return '?' character.

The sensor (see Figure 1) will be realized by switches placed on the Arcom board.

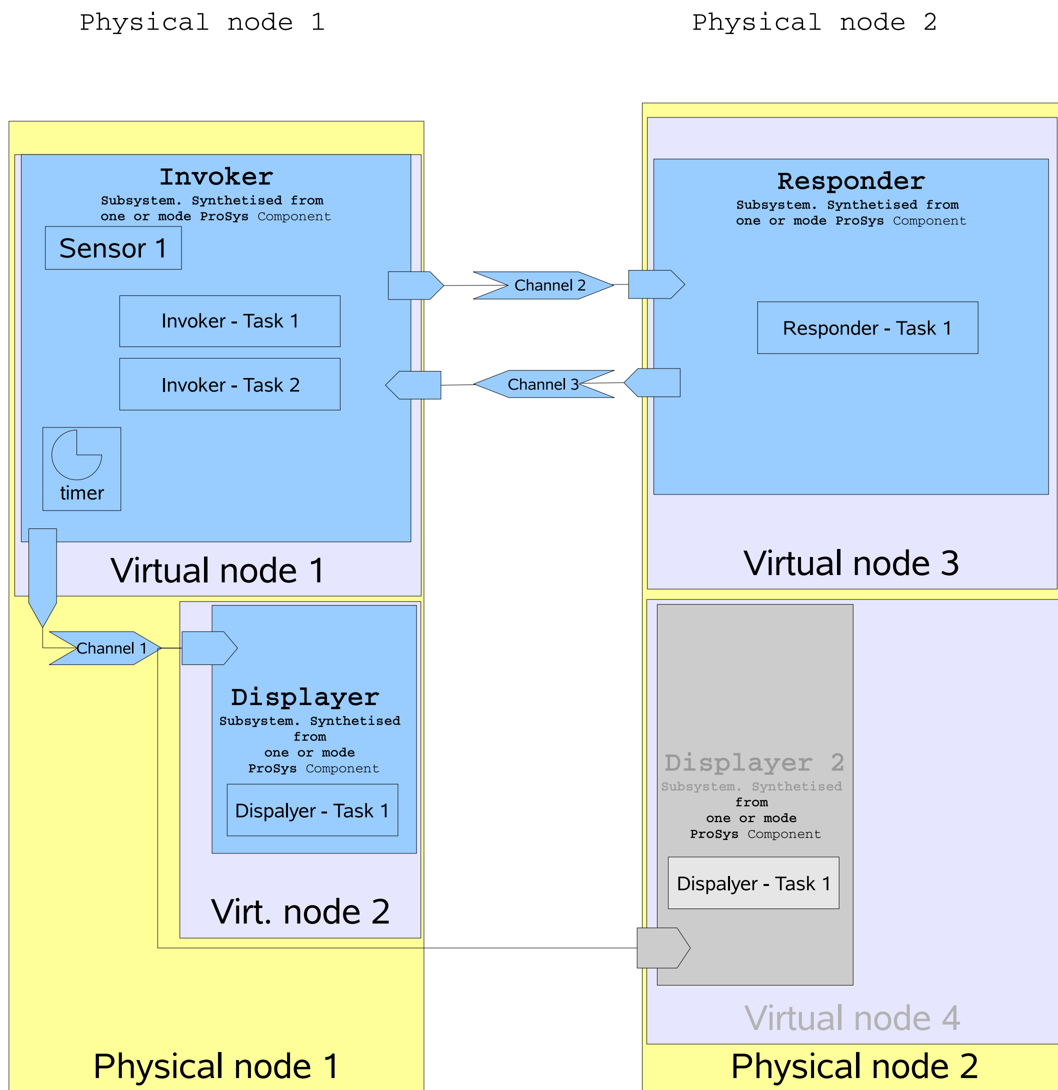

Figure 1: Sample application

7.4 Requirements to application

Application will be written in the "task style". There will be set of concurrent tasks running on each node.

The application should be simple, but as similar as possible to some real use of embedded RT systems.

7.5 Channels in application

There was developped simple implementation of the channels for the purposes of this thesis.

7.5.1 Design of the transfer mechanism

We have tried to use very general aproach to design prepared for demo aplication, but for some architectures of hardware could be complicated this architecture realized. But this design should cover wide spectrum of applications and hardware platforms.

In addition to the logical separation to connectors and channels, demo implementation of the channels introduce also concept of channel front-ends and channel back-ends. Channel is logically composed from front-ends, backends, and channel attributes.

For every channel and every physical node, where at least one output connector associated with this channel exists will be created channel frontend on every other physical node, where at least one input connector associated with this channel exists. This frontend holds the information about adress of remote backend of the channel, where the data should be delivered.

Similaryly, for every channel and every physical node, where at least one input connector associated with this channel exists will be created channel backend on every other physical node, where at least one output connector associated with this channel exists will be created channel backend. Backend holds information about all input connectors, associated with this channel, which are placed in this physical node and is responsible for receiving messages.

As was already introduced in part about time analysis of the channels, there exist a delivery task on every physical node and receiver task for every remote frontend of each channel.

All the mechanism is shown on figure 2.

The whole message transfer mechanism will be composed from following parts:

1. Sender task writes data to input buffer and wake up delivery task.
2. Delivery task take care about delivering all the data to the destination.
3. For the ethernet connections, the receiver task will receive the data and writes it into outpus buffer of the channel.
4. Receiver task is signaled (via semaphore) that next data are ready.

The way, how the code implementing channels and connectors is done was organised to allow to add another types of comunication easyly. It consist of

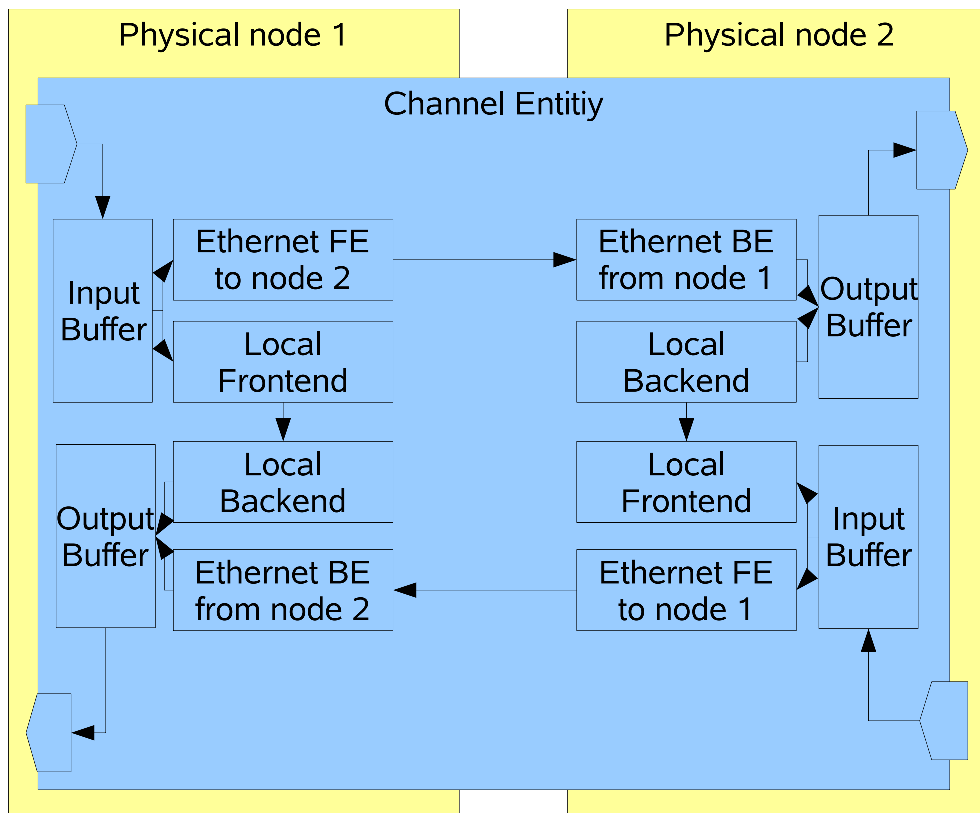

Figure 2: Channel design. Two physical nodes connected via a channel

Connector Implements connectors. Entity of connector holds some atributes and wrapper calling of channel functions

Channel Implements channels wrapper. This code is not necessary change, if the particular implementation (for example for other underlied system) changes

Local transfer implementation Implementation of local data transfer.

Ethernet tranfer implementation Implementation of local data transfer.

7.5.2 Local Data Transfer Implementation

Local implementation is very simple. It consist of input buffer, output buffer and data transer function. This function simply copy the data from input buffer to putput buffer.

More compicated is synchronization of the readers and writers. Channel holds information about all the readers. After data arives, semaphore responsible for signaling particular reader is posted. When reader reads data, he posts semaphore responsible for signaling writer.

7.5.3 Ethernet Data Transfer Implementation

There are used sockets for the channel implementation.

In the time of system start-up, TCP-IP connection between each frontend and asociated backend is created.

Synchronization of the readers and writers is very similar to local one.

At first all frontends are created, than the systems is trying to connect backends to aproprite backends on other physical nodes. If the connection can not be established imediately, backend will sleep one second and then try it again. This allows situation, when physical nodes are not started in exactly the same time.

7.5.4 Channel Testing Application

There was prepared extra application to show channel impelementation. It schema is of the figure 3

7.6 Code of the VN

This is an example, how the code of one task o some virtual node could look like (at least in my sample application).

Code of the vitrual node is composed of tasks. This tasks are defined by one C function of given type. The type of the function allows to pass any data imside the task and also return pointer to any data.

The wrapper function task_run() will return in time, when all the tasks from this virtual node finish its work.

```
/* one task of virtual node */
void* vnode_writer_task_1(void * unused){
    char c;
```

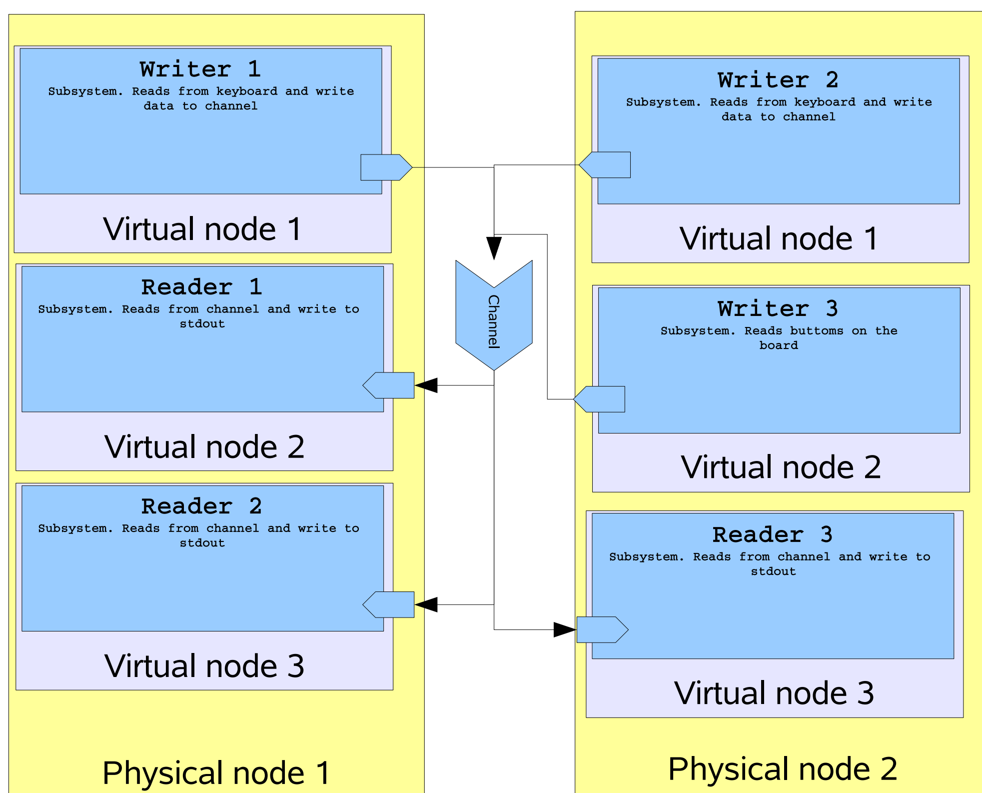

Figure 3: Application created to show channels functionality

```
    c = sensor_reader();
    while (c != ' '){
        printf("WRITER: writing to OUTPUT_0 (channel %d) '%c'\n",(WRITER_OUTPUT_0.
        fflush(stdout);
        connector_write_triggered(WRITER_OUTPUT_0, (void *) &c);
        c = sensor_reader();
    }
    connector_write_triggered(WRITER_OUTPUT_0, (void *) &c);
    printf("WRITER: writing to OUTPUT_0 (channel %d) '%c'\n",(WRITER_OUTPUT_0.chan
    fflush(stdout);
    return NULL;
}

#define WRITER_NR_TASKS 1

/* routine, which will run virtual node. */
void * vnode_writer_run(void * unused){
    int i;
    task_handle_t tasks[WRITER_NR_TASKS];
    tasks[0] = task_run( vnode_writer_task_1 );
    for (i = 0; i < WRITER_NR_TASKS; i++){
            task_wait(tasks[i]);
    }
    return NULL;
}
```

7.7 Notes

The planed application is more complicated then I have original planed, but could demonstrate many things and may be will be better for demonstrative purposes.

The application may change a little during implementation.

References

[1] Tomáš Bureš, Jan Carlson, Ivica Crnković, Séverine Sentilles, Aneta Vulgarakis *ProCom – the Progress Component Model Reference Manual*

T P