MyriaChap Implementation Details

Author	Freek van Polen, Almende B.V. Rotterdam, the Netherlands
Version	3.x
Status	Concept
Date	19 October 2011

Table of Contents

1. Introduction

MyriaChap is a WSN platform for creating evolving and evolvable applications. The main functionality is that it allows individual nodes to change their behaviour at runtime. This is achieved by breaking down functionality into small blocks, called components, each implementing some small function. These components are then combined by connecting them through subscriptions, causing one component to take as input and operate on the output of another component. Components can be created deleted, and recombined at runtime by the node itself, thus allowing autonomous adaptation of nodes.

The main rationale behind the implementation of MyriaChap is for the wireless sensor network to be self-organizing, not only on a network level, but on an application level as well. Typical WSN applications require large networks with many nodes, that have to monitor many different parameters in an environment. Having to program each of the nodes of the network individually to exhibit the desired behaviour is undesirable as it causes installation of networks to be a tedious task, and causes the behaviour of networks to be static. Self-organization both on the network level and the application level is thus a very important feature of wireless sensor networks, and it necessitates a middleware that allows nodes to adapt their behaviour.

In MyriaChap functionality is divided in small *components*, that implement a certain function, and which can be created and deleted dynamically at runtime. Compositions of components are constructed by letting components *subscribe* to each other, causing a subscribing component to receive input from another component. This way it will take the output of the other component as input for whatever its own operation is. An example is a component that calculates an average value every ten minutes that subscribes to a component that takes a temperature reading every minute. The resulting behaviour of the node as a whole is that it generates an average temperature every ten minutes. These subscription can also be changed during runtime, allowing the node to for instance delete the temperature component, create a humidity component, and have the averaging component subscribe to the humidity component.

An additional feature of MyriaChap is that it takes care of scheduling the operation of the components, as well as the passing of control between components. Components communicate with each other by sending messages; simple data structures containing a subject and a content. When a message is being delivered to a component, the MyriaChap *scheduler* calls the component's inbox function, and the component can subsequently decide what to do with the message. After it is done processing the message, it passes control back to the scheduler, which will then proceed to deliver the next message in queue. Messages can be scheduled to be delivered with a certain delay, allowing periodic operation of components. Examples of this mechanism at work are the temperature component notifying the subscribed averaging component that a new value is ready: it sends a message to the averaging component with the correct subject and the data in the content. The averaging component gets control and can for instance copy the data to a local storage. The mechanism is also used by components to schedule themselves for periodic operation: the temperature component needs to run again in one minute time, so it can send a message to itself which is scheduled to arrive after one minute. The averaging component would use the same mechanism to calculate an average value every ten minutes. Note that any single component can have different messages scheduled to be delivered to it, each with different subjects, and so causing different behaviour by the component.

In this document MyriaChap is explained in more detail. We will first take a look at the main data structures that make up MyriaChap. While an object oriented programming language might have been the obvious choice for implementing MyriaChap, it was implemented in C so that it can be run on wireless sensor nodes which typically have very limited hardware features. This causes the code to look somewhat contrived, but it keeps the memory and RAM footprints very small.
Next we will take a look at how one can build ones own components. While MyriaChap provides all the necessary infrastructure and data structures required, it does not offer any predefined components.
We wil then proceed with an application example, showing how an application can be built making use of MyriaChap.
Finally, we will consider the scheduler internal to MyriaChap, as it will require some tailoring to the specific operating system MyriaChap is run on. We will include implementation details and pointers for scheduler interfaces for MyriaNed and TinyOS.

2. MyriaChap Data Structures

Since MyriaChap was implemented in C, a number of data structures are its basic building blocks. One could think of these structures as classes in an object oriented programming language, but the implementation is a bit awkward. The way to give a structure in C a function that can be called from outside and that operates in the context of that structure, the way a function in an object operates in the context of that object, is to give the class a pointer to a function as a member, and to call the function pointed to by that member of the structure with the structure itself as an argument. Another feature from object oriented programming that is loosely imitated in MyriaChap is that of inheritance. This is achieved by giving the abstract class a void pointer as member, that can be set to point to the inheriting object. The inheriting object can then contain a pointer back to the component. As all functions that are specific to the inheriting class can know to what class they belong, they can simply take an argument of abstract class type and cast its void pointer to the inheriting class, and so operate in the context of the inheriting object.

2.1. Component Data Structure

The Component data structure contains the basic elements that any component should be built up of.
==========

```
struct Component_Struct {
        void (*processMessage) (void *schedulee, void *context, void *scheduler);
        struct Subscription_Struct **outputs;
        Settings settings;
        void *inheriter;
};
typedef struct Component_Struct Component;
```

As can be seen, the members of the Component struct are:

- *processMessage:* a pointer to a function that is called by the scheduler when a message is to be delivered to this component. The behaviour, and thus the identity of the component, is determined by this function. The pointer needs to be set when the component is created.
- *outputs:* a two dimensional array of subscriptions. Every component has zero or more output channels, to each of which zero or more components may be subscribed. These subscriptions are stored in this two dimensional array.
- *settings:* a structure containing a number of settings of the component. Includes as most interesting members *run_interval*, which controls how often the component needs to run (if the *run_interval* is 0, the component does not run periodically) and *nrOfOutputs*, which denotes the amount of output channels the component has, and which is static.

2.2. Subscription Data Structure

The Subscription data structure is used by component to subscribe to each others output.
==========

```
struct Subscription_Struct {
        uint8_t outputNumber;
        Component *subscriber;
        uint16_t subject;
        struct Subscription_Struct *next;
};
typedef struct Subscription_Struct Subscription;
```

As can be seen, the memmbers of the Subscription struct are:

- *outputNumber:* the index of the output channel that is subscribed to.
- *subscriber:* a pointer to the subscribing component, so that it can be notified when new data is available for the relevant output channel.
- *subject:* the subject that the subscribing component wants notification messages to have. Using this subject, the subscribing component can distinguish from which of its subscriptions a received

message originated.

2.3. Message Data Structure

The Message data structure is used by component to communicate with and schedule each other.

```
struct Message_Struct {
        uint16_t subject;
        void *  content;
};
typedef struct Message_Struct Message;
```

As can be seen, the members of the Message struct are:

- *subject:* a number denoting the subject of the message, much like the subject of an e-mail message. The receiving component will determine what to do with the message using the subject of the message.
- *content:* a pointer to the content of the message. Depending on the way the receiving component handles messages with this subject, it may or may not try to access the content of the message. The sending component is obliged to maintain the content until the receiving component has handled the message. The content can be a NULL pointer, if the specific type of message requires no content.

2.4. Memory Management: boxStorage

On embedded devices, the use of the limited amount of memory needs to be handled carefully, as it can cause many problems. One can easily use up too much memory, causing an application to crash, and even if managed well, fragmentation due to excessing allocation and freeing of memory may cause issues. To keep these problems to a minimum, MyriaChap can take care of at least part of the memory management. MyriaChap does this by allocating a fixed amount of memory for Components, Subscriptions and Messages, which it will make available to the application. This mechanism will make sure that one will never run out of memory at run time, since the required amount of memory should have been allocated during startup. Also, the same memory is reused when old components are removed and new components are made, making sure memory does not get fragmented.

The required functions are defined in boxStorage.h. It contains a function for initializing the required amount of memory, as well as functions for retrieving an empty Component, Subscription or Message struct and returning one.

```
uint8_t initStorageSpace (uint8_t msgPoolS, uint8_t comPoolS, uint8_t subPoolS);

Message *getEmptyMessage ();
void returnMessage (Message *msg);

Component *getEmptyComponent (uint8_t nrOfOutputs);
void returnComponent (Component *com);

Subscription *getEmptySubscription ();
void returnSubscription (Subscription *sub);
```

3. Building Components

Implementing a component involves a number of steps. First, if the component has need of more memory or functions that the base Component struct offers, one can define a new struct specifically for this component. This new struct should always contain a pointer to a base component, and using the base component's *inheriter* field point to the new struct, an inheritance mechanism emerges. Second, define constructor and destructor functions for the components. These function are used to (de)allocate any memory that might be required, to initiate any hardware modules (sensors), etc. Finally, define an inbox function for the new component, that essentially defines the behaviour, and therefor the identity of the new component. The inbox function determines how the component will react to messages with different subjects. Typically, there is only a few subjects which need handling, as many subjects have a standard way of handling which are already defined in *chapStdLib*. Below we create a component for a temperature sensors as an example.

3.1. TemperatureComponent Struct

The base Component struct does not offer any memory for the component to use, but our temperature component will need some to store a reading it just took. After taking a reading the component will immediately notify any subscribed components, so it only needs to store one reading at a time. Since this requirement is shared by pretty much any component whose job it is to read out a sensor, we will give the struct a more general name.
==========

```
struct SensorComponent_S {
	Component *C;
	int16_t value;
};
typedef struct Sensor_Component_S SensorComponent;
```

3.2. *TemperatureComponent (Con-/De-)structor*

During construction of the component we need to up the Component structure, as well as the required (semi-) static parameters of the Component. The structure of the component includes linking the SensorComponent struct to a base Component struct and vice versa, and setting the inbox function of the base component to the inbox function of this component that we will define later. The parameters include the amount of output channels the component has and the frequency with which it has to run. Additionally, we may need to initiate the hardware here.
==========

```
SensorComponent *getTemperatureComponent (uint32_t run_interval_sec, uint16_t run_interval_msec) {
	SensorComponent *T = malloc(sizeof(SensorComponent));
	if(T == NULL)
		return NULL;

	T->C = getEmptyComponent(1);
	if(T->C == NULL) {
		free(T);
		return NULL;
	}

	T->C->inheriter                         = T;
	T->C->processMessage                    = &inboxTemperatureComponent;
	T->C->settings.run_interval.sec         = run_interval_sec;
	T->C->settings.run_interval.msec        = run_interval_msec;
	T->C->settings.nrOfOutputs              = 1;
	return T;
}

void removeTemperatureComponent (TemperatureComponent *T) {
	returnComponent(T->C);
	free(T);
}
```

3.3. TemperatureComponent Inbox

In the inbox function of any component we first need to gather the TemperatureComponent itself and the message from the arguments. (Remember that C is not an OO programming language, and the function is not called in the context of any object. Rather, the object is given as an argument of the function.) After the arguments have been gathered, the message can be evaluated and proper action can be taken. As mentioned earlier, we need not implement the response to a lot of standard messages, and can rather pass the message to a standard inbox function if the message subject is not recognized.
==========

```
void inboxTemperatureComponent (void *schedulee, void *context, void *scheduler) {
	SensorComponent *me = (SensorComponent *) ((Component *) schedulee)->inheriter;
	Message *msg = (Message *) context;

	switch(msg->subject) {
	case RUN:;
		// get a temperature measurement
		me->value = mnSHT21GetTemperature();
		// notify other components a new value is available and where
		notifySubscribers(me->C, 0, &(me->value));
		// schedule self to run again according to run_interval
		scheduleSelf(me->C, RUN, MEDIUM_PRIORITY);
		break;
	defaultcase:
	default:
		// if not recognized, pass the message to the stdInbox
		stdInbox(schedulee, context, scheduler);
		break;
	}
}
```

3.4. chapStdLib Functions

The chapStdLib header file offers a number of standard functions that can be used in the application or by components themselves. In particular it offers a standard inbox function that recognizes a number of standard message subjects. Using this standard function relieves one of the need to implement many configuration messages in new components.
==========

```
/**
 * Requests the run interval from another component.
 * @param me   The requesting component
 * @param him  The component whose run interval is requested
 */
void getRunInterval (Component *me, Component *him);

/**
 * Increases the run interval of a component, thereby decreasing the frequency with which it runs.
 * @param Co   Component whose run interval to adjust
 * @param sec  Amount of seconds to add to the run interval
 * @param msec Amount of milliseconds to add to the run interval
 */
void adjustRunIntervalUp (Component *Co, uint32_t sec, uint16_t msec);

/**
 * Decreases the run interval of a component, thereby increasing the frequency with which it runs.
 * @param Co   Component whose run interval to adjust
 * @param sec  Amount of seconds to remove from the run interval
 * @param msec Amount of millisecond to remove from the run interval
 */
void adjustRunIntervalDown (Component *Co, uint32_t sec, uint16_t msec);

/**
 * Copy component settings to a component. Used to give a component new settings.
 * @param S    The settings to copy to the component.
 * @param to   The component to which to the copy the settings
 */
void copySettings (Settings *S, Component *to);
```

```
/**
 * Schedule a component for execution according to the component's run interval.
 * @param Co          Component to schedule
 * @param subject     Subject of the message to send
 * @param priority    Priority of the message
 */
void scheduleSelf (Component *Co, uint16_t subject, uint8_t priority);

/**
 * Schedule a component for immediate execution, irrespective of its run interval.
 * @param Co          Component to schedule
 * @param subject     Subject of the message to send
 * @param priority    Priority of the message
 */
void scheduleSelfNow (Component *Co, uint16_t subject, uint8_t priority);

/**
 * Schedule a component for execution in the next round (MyriaNed only).
 * @param Co          Component to schedule
 * @param subject     Subject of the message to send
 * @param priority    Priority of the message
 */
void scheduleSelfNextRound (Component *Co, uint16_t subject, uint8_t priority);

/**
 * Connect two components.
 * @param inputSide   Component that is on the receiving end of the connection
 * @param outputSide  Component that is on the sending end of the connection
 * @param outputNr    Nr of the output channel of the outputSide component that should be used
 * @param subject     Subject of the notification message that outputSide should send
 */
void connectComponents (Component *inputSide, Component *outputSide, uint8_t outputNr, uint16_t
subject);

/**
 * Notifies all subscribed components that new data is available, and where to get it.
 * @param Co          Component that has new data available
 * @param outputNr    Nr of the output channel for which new data is available
 * @param dataPtr     Pointer to the new data. Co is obliged to leave the data here until all
subscribers have had a chance to retrieve it
 */
void notifySubscribers (Component *Co, uint8_t outputNr, void *dataPtr);

/**
 * The standard inbox function which takes care of standard messages.
 * @param schedulee   The scheduled component, or recepient of the message
 * @param context     The message itself
 * @param scheduler   The scheduling component, or the sender of the message
 */
void stdInbox (void *schedulee, void *context, void *scheduler);
```

The message subjects that are handled by *stdInbox*:

```
=========================================================

#define ORDER_SUBSCRIBE
#define SET_SETTINGS
#define SUBSCRIPTION
#define GET_RUN_INTERVAL
#define SET_RUN_INTERVAL
#define ADJUST_RUN_INTERVAL_UP
#define ADJUST_RUN_INTERVAL_DOWN
```

4. Building an Application

In this chapter we will use an example application to demonstrate how an application can easily be built making use of predefined component types.

4.1. Predefined Components

We assume that the following component types have been defined and are available for use.

- *TemperatureComponent:* A component that periodically takes a temperature reading from a temperature sensor, has one output channel, and outputs whatever data it gathers over this channel.
- *AveragerComponent:* A component that periodically calculates the average and variance of any values it has received as input since the previous calculation. Upon being notified that new input is available, it copies the value to an internal storage area. When it is time to calculate the average, it does so using the values that have been stored internally, and outputs the average over the first output channel, and the variance over the second output channel.
- *GossipComponent:* A component that is quite specific to the data diffusion library used in MyriaNed, and that takes care of diffusing data through the network. It takes any data it receives as input, labels it, and diffuses it through the network. This component has not output channels.

4.2. Includes, Defines, Global Variables

Since MyriaChap and the predefined components take care of most of the functionality of the application, the main application file itself can remain very compact. At the top of the file we will have to include the MyriaChap header file, as will as the header files containing the function prototypes and structure declarations of the component types.
Additionally, we will declare the components that will be used as global variables, so they can be used anywhere in the application. If one wishes to make a truly adaptive application, these variables can be of void type, and merely be placeholders for components that may be created during runtime.
==========

```
// MyriaChap related includes
#include "myriaChap.h"
#include "timedStateComponents.h"
#include "sensorComponents.h"
#include "averagerComponent.h"

SensorComponent              *TemperatureComp;
AveragerComponent            *AveragerComp;
GossipComponent              *TemperatureItem;
```

4.3. Application Initialization

Before the application can start running, MyriaChap needs to be initialized, as well as the components instantiated and configured. While the instantiation and configuration of the components could also be done at runtime, the initialization of MyriaChap can not be postponed to after startup.
==========

```
// Initialize MyriaChap storage space
initStorageSpace(30, 14, 10);
// Initialize MyriaChap scheduler interface
initSchedulerInterface(FRAME_TIME);

// Obtain a Temperature Component with runtime of 60 secs and 0 msecs
TemperatureComp       = getTemperatureComponent(60, 0);
// Obtain an Averager Component with runtime of 600 secs and 0 msecs
AveragerComp          = getAveragerComponent(600, 0);
// Obtain a Gossip Component and provide it with a label
TemperatureItem       = getGossipComponent(TEMPERATURE_SHT21 * 256 + nodeID);

// Let the Averager Component subscribe to the Temperature Component
```

```
connectComponents(AveragerComp->C, TemperatureComp->C, 0, DATA_AVAILABLE);
// Let the Gossip Component subscribe to the Averager Component
connectComponents(TemperatureItem->C, AveragerComp->C, 0, DATA_AVAILABLE);

// execute the tasks to get the subscriptions done
executeTasks();
// let the Temperature and Averager components schedule themselves to get things going
scheduleSelf(TemperatureComp->C, RUN, MEDIUM_PRIORITY);
scheduleSelf(AveragerComp->C, RUN, MEDIUM_PRIORITY);
```

4.4. Application Main Routine

With the configuration of components done, most of the work is out of the way. What remains to be done depends on the operating system on which MyriaChap runs, and whether MyriaChap has a software or hardware timer available to work on. If such a timer is available, and MyriaChap has been initiated to use it, nothing else needs to be done, and the components will take care of everything. If no timer is available, MyriaChap will need to receive a time tick periodically, so it knows how much time has passed.
==========

```
// Poke the scheduler to tell it time has passed by
pokeScheduler();
// Let MyriaChap execute any tasks that have become eligible for execution
executeTasks();
```

5. SchedulerInterface

There are two mandatory functions that SchedulerInterface should provide to the application and to components; *initScheduler* and *scheduleComponent*.

initScheduler takes care of initializing the Scheduler in the correct way, basically passing three function pointer to it; a setTimer, getTimer, and notification. The specifics of the underlying operating system are hidden from the Scheduler, and thus it will use these functions the same way regardless of the implementation. It is up to the specific SchedulerInterface implementation to provide the correct functions so that the whole will work.

scheduleComponent allows an application or a component to the send a message to an other or the same component. This function must turn the message into a Task and offer it to Scheduler. Sending a message requires a lot of parameters, and because of that chapStdLib implements some functions that will send a message with preset settings. It is advisable to use these functions offered in chapStdLib as much as possible.

==========

```
/**
 * Send a message to a component
 * @param F  Pointer to a function to be executed, typically the inbox function of the receiving
              component
 * @param C1  Pointer to the component that should receive this message. Will be delivered as
              argument "schedulee" to function F.
 * @param C  Pointer to the message to be sent. Can be NULL. Will be delivered as argument "context"
              to function F.
 * @param C2  Pointer to the component that is sending this message. Can be NULL. Will be delivered
              as argument "scheduler" to function F.
 * @param p  Priority of this message. A higher priority will cause the message to take precedence
              over messages with a lower priority but the same due time.
 * @param t  Pointer to a Time struct denoting the delay with which this message must be delivered.
              Can be NULL, which means it should be delivered directly, and so the message will
              become a Task instead of a TimedTask.
 * @param id  The id of the message, that will allow components or the application to review whether
              a specific message is already scheduled or not.
 * @return  void
 */

void scheduleComponent (void (*F) (void *schedulee, void *context, void *scheduler),
                        void *C1, void *C, void *C2, uint8_t p, Time *t, uint8_t id) {

}
```

5.1. MyriaNed SchedulerInterface

The MyriaNed platform does not (in the current stable version) offer timers for the application to use, instead passing control to the application periodically, at a fixed interval. This interval is called the FrameTime. The SchedulerInterface hides this fact from the Scheduler. Every Frame, the application calls *pokeScheduler*, which gives a a tick to the Scheduler for FrameTime. The Scheduler will then see if this is enough for a TimedTask to be lined up for execution. (In fact, the Scheduler will line up all TimedTasks that have become eligible since the previous time tick.
Next, the application calls *executeTasks* which will execute all tasks in the TaskQueue. Any new Tasks that get inserted into the TaskQueue during execution of a Tasks will automatically also be executed by *executeTasks* in that same round.
==========

```
Time FrameTime;

void pokeScheduler () {
       giveTimeTick(&FrameTime);
}

void executeTasks () {
```

```
        Task *toExecute = retrieveTask();
        while(toExecute != NULL) {

                toExecute->function(toExecute->schedulee, toExecute->context, toExecute->scheduler);
                returnMessage(toExecute->context);
                taskCompleted(toExecute);

                toExecute = retrieveTask();
        }
}
```

5.2. TinyOS SchedulerInterface

In case the operating system offers timers for the application to use, as is the case in TinyOS, the following SchedulerInterface implementation can be used. Pointers to the *setTimer* and *getTimer* functions are passed to the Scheduler, so it is now able to work with these timers. The SchedulerInterface implements an Interrupt Service Routine that captures the event that a timer generates. When this happens, it notifies the scheduler by calling *giveTimeTick*, the amount of time spent is then equal to the value that Scheduler last set the timer to. This will cause Scheduler to line up the first TimedTask in the TimedTaskQueue for execution, and to set the timer to go off when the next TimedTask is due.
SchedulerInterface can use the same while loop to execute all tasks that are lined up as described above. The main difference is that it is not a function that is called periodically, but rather set as the notification function in Scheduler. This notification function is called by Scheduler whenever a Task is added to an *empty* TaskQueue.
==========

```
Time currentTimer;

void setTimer (Time *t) {
        currentTimer.sec  = t->sec;
        currentTimer.msec = t->msec;

        // set OS-timer
}

Time *getTimer () {
        // retrieve OS-timer

        return &currentTimer;
}

void timer_isr () {
        giveTimeTick(&currentTimer);
}

void taskReady () {
        Task *toExecute = retrieveTask();
        while(toExecute != NULL) {

                toExecute->function(toExecute->schedulee, toExecute->context, toExecute->scheduler);
                returnMessage(toExecute->context);
                taskCompleted(toExecute);

                toExecute = retrieveTask();
        }
}
```