COMPOSING AND CONNECTING DEVICES IN ANIMAL TELEMETRY NETWORK

by

ASHWIN KRISHNA

B.Tech., Amrita School of Engineering, Bangalore, 2012

A REPORT

submitted in partial fulfillment of the requirements for the degree

MASTER OF SCIENCE

Department of Computer Science
College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

2016

Approved by:

Major Professor
Venkatesh P Ranganath

Abstract

As the Internet of Things (IoT) continues to grow, the need for services that span multiple application domains will continue to increase to realise the numerous possibilities enabled by IoT. Today, however, heterogeneity among devices leads to interoperability issues while building a system of systems and often give rise to closed ecosystems. The issues with interoperability are driven by the inability of devices and apps from different vendors to communicate with each other. The interoperability problem forces the users to stick to one particular vendor, leading to vendor lock-in. To achieve interoperability, the users have to do the heavy lifting (at times impossible) of connecting heterogeneous devices.

As we slowly move towards system-of-systems and IoT, there is a real need to support heterogeneity and interoperability. A recent effort in Santos Lab developed Medical Device Coordination Framework (MDCF), which was a step to address these issues in the space of human medical systems. Subsequently, we have been wondering if a similar solution can be employed in the area of animal science.

In this effort, by borrowing observations from MDCF and knowledge from on-field experience, we have created a demonstration showcasing how a combination of precise component descriptions (via DSL) and communication patterns can be used in software development and deployment to overcome barriers due to heterogeneity, interoperability and to enable an open ecosystem of apps and devices in the space of animal telemetry.

Table of Contents

List of Figures

List of Tables

Acknowledgements

First and foremost, I would like to thank, Prof. Venkatesh P Ranganath, for serving as my major professor. It was a great learning experience and was a pleasure working under him. I would also like to thank Dr. John Hatcliff and Dr. Bob Larson for serving on my supervisory committee. A special thanks to Dr. Eugene Vasserman for believing in my ability and providing me with an opportunity to be part of SAnToS Laboratory.

I am extremely grateful to my family - both my parents, my brother, as well as my cousins Anil and Ashok - have been a source of great support and inspiration, and I would not have made it nearly as far as I have without them.

Chapter 1 – Current State on Farms

The purpose of a telemetry system is to reliably and transparently convey measurement information from a remotely located data generating source to users located elsewhere for monitoring. Animal Telemetry is a process of tracking, monitoring and recording either the physiological properties of an animal or the environmental properties of the location in which the animal is placed, by the use of telemetry systems.

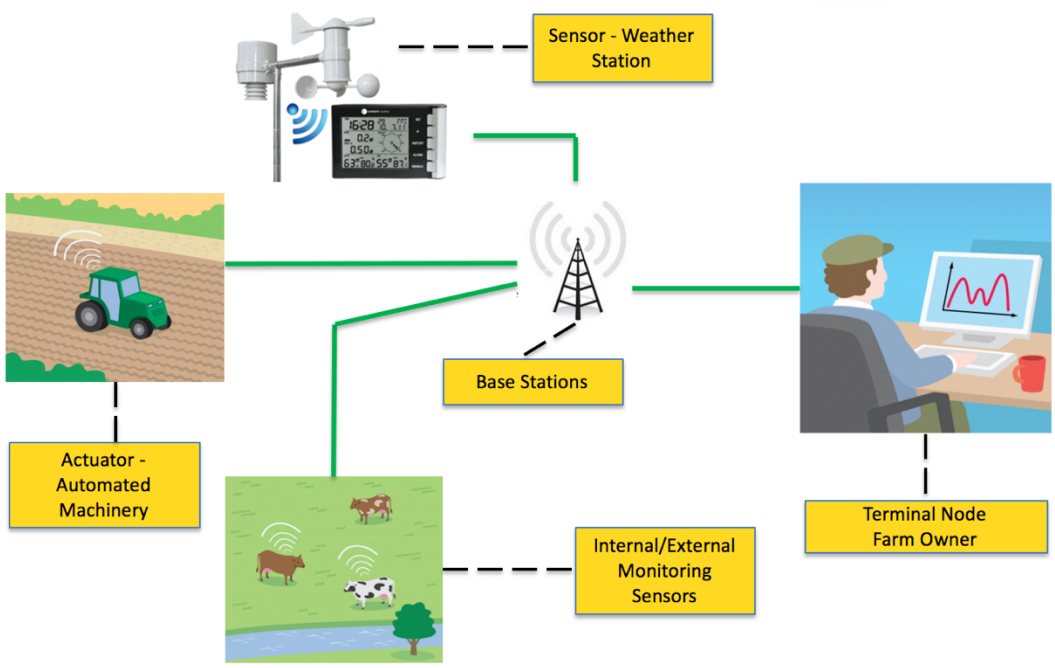

Figure 1.1 Devices that could be found in a farm.

Typically, sensors act as data sources. Sensors are available in a variety of shapes and sizes to accommodate species requirements. Some of the sensors are implantable while some are not. For example, a "Rumen Bolus" is a sensor that is placed in the rumen of the cattle to measure the core body surface temperature, while a "Fever Tag" is a sensor that is placed in the ear canal of the cattle to measure the body temperature. The implantable sensors mostly capture the physiological properties of the animal. The non-implantable sensors along with capturing

physiological functions can also obtain environmental parameters. These non-implantable sensors that capture the environmental parameters are usually set on wild animals that are free to roam. In this project, we would be focusing on domesticated farm animals which mean that the sensors used for animal telemetry are for capturing physiological parameters of the animal. As the movement of animals is restricted by a farm boundary, the environmental parameters are obtained using wired or wireless sensors that are deployed somewhere in the farm. For example, a "Weather Station" sensor could be used measures wind speed, atmospheric pressure and the air temperature in the farm. Figure 1.1 give a pictorial view of the different devices that could be found in a farm.

The data transmitted by the sensors are captured by the receivers and are passed to a computer for monitoring. There can be multiple transmitters (aka base stations) in-between to boost the signal in case of large farms. These receivers and transmitters do not play any significant role in the context of this project; hence they would be ignored from the rest of the discussion. The farm owner uses a monitoring software (provided by the manufacturer of the sensors) on his computer to capture and visualize the data. Actuators are another class of devices found in such settings. This class of devices performs actions based on input commands. For example, an actuator that can automatically open and close gates of the farm, an automated air cooler that switches on and off to regulate the temperature inside cattle shed.

1.1 Issues with the Current State

A large number of these devices (especially sensors) are usually deployed in a farm. It would become increasingly difficult for a farm owner to keep track of all the devices as their number of increases. There are several reasons for it which are mentioned below

1. Limited functionality of the monitoring software – It is common that farm owners have to tend to various activities on the farm while wanting to keep tab on the health of their cattle. For example, when the temperature of calf rises beyond the normal threshold, they would prefer to be notified (just as in a hospital) as opposed to periodically keeping track of the temperature of 10s and 100s of calves. While such notification can be automated, it cannot be automated if the monitoring software does not support any alerting mechanism or communicate with an alerting system available on the farm.
2. Interoperability problems among devices – Interoperability is concerned with the ability of systems to communicate – and it requires that the communicated information can be understood by the receiving system. So, if the farm owner wants to automate the routine mundane task of regulating the temperature inside the cattle shed using air cooler depending upon the weather station data, he may not be able to so because of the two systems not being interoperable.
3. Interoperability problems among software – The monitoring software are tightly tied to devices from specific vendors and are not interoperable with other devices. So, if the farm owner likes software of vendor "A" and a sensor of vendor "B" and wants to monitor device from B with software from A, he may not be able to do so.
4. Too many monitoring applications – Due to interoperability problems, as the heterogeneity increase the number monitoring applications he would have to simultaneously run increases and is often a headache to monitor all at once.
5. Longer setup and deployment time – Each of the hardware and software components have their own requirements and setup procedures, and since they differ from vendor to vendor. Typically, farm owners are non-experts when it comes to internal workings of

devices and software. This can act as a non-trivial barrier in deploying, assembling, and configuring a monitoring system leading to longer set and deployment times.

Given all these problems that exists with animal telemetry, we will take a look at some of the concepts and implemented solutions that are available today that promise to break the heterogeneity and interoperability barriers.

1.2 Internet of Things

The term **Internet of things** (**IoT**) refers to a network of physical devices, vehicles, buildings and other items - embedded with electronics, software, sensors, actuators, and network connectivity that enable these objects to collect and exchange data [1].

The term Internet of Things was first coined by Kevin Ashton in 1999 in the context of supply chain management [2]. However, over the past decade, the definition has been more inclusive covering wide range of applications like healthcare, utilities, transport, etc. [3]. Thanks to rapid advances in technologies, IoT is opening tremendous opportunities for a large number of novel applications that promise to improve the quality of our lives.

As increased interest for IoT spread across various sectors of the industry, in one of the sectors, especially healthcare, it leads to a new paradigm of medical systems called the Medical Application Platforms (MAP) [4]. A MAP is a safety- and security- critical real-time computing platform for (a) integrating heterogeneous devices, medical IT systems, and information displays via a communication infrastructure and (b) hosting application programs (i.e., apps) that provide medical utility via the ability to both acquire information from and update/control integrated devices, IT systems, and displays.

Figure 1.2 illustrates the clinician's view of a clinical-based MAP. A communication infrastructure connects medical devices that are communications enabled (e.g. via Bluetooth,

USB, or Ethernet) as well as other information systems such as a patient electronic medical record (EMR) and drug dosing databases. A device database records the unique identifiers and drivers/interfaces for devices that have been pre-approved for connection to the framework. The app execution environment would typically include a library of apps written by experts and (possibly, if it implements medical device functionality) approved by appropriate regulatory authorities (e.g., the US Food and Drug Administration). A clinician desiring a particular medical system behavior chooses an appropriate app from the library. Each app contains a list of device types and associated device capabilities that are required to carry out a medical system activity. During the app initialization phase, the app execution environment attempts to acquire devices that satisfy the device requirements of the app and are currently connected to the communication infrastructure. After a complete set of required devices has been selected and confirmed as available, app execution begins. App execution may proceed without intervention, or may stop to receive input from the clinician.

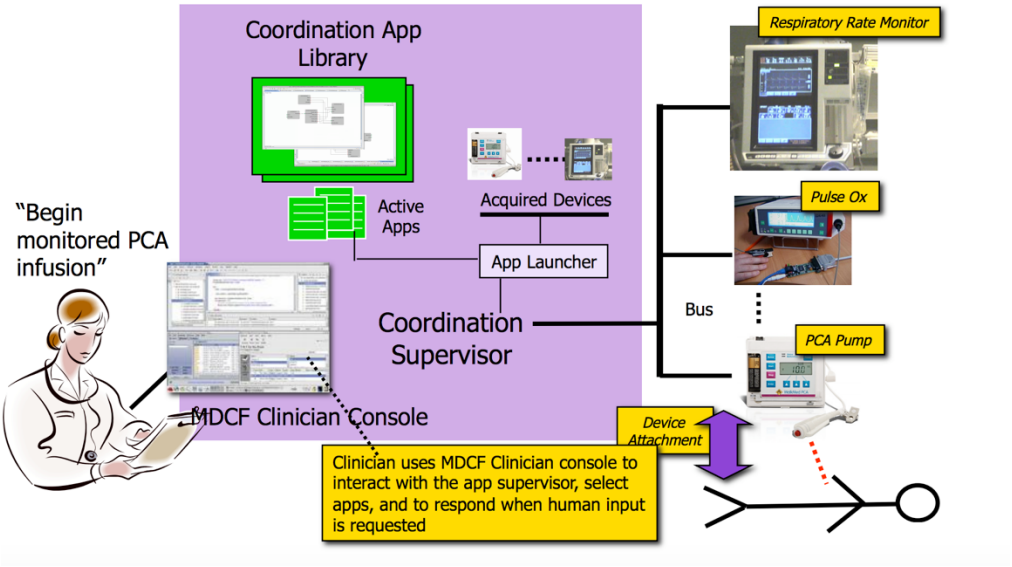

Figure 1.3 Clinician's View of a Medical Application Platform

The Medical Device Coordination Framework (MDCF) [5] is an open source Medical Application Platform (MAP) developed jointly by Kansas State University and the University of Pennsylvania, which facilitates interoperability between heterogeneous medical devices.

Based on the perceived benefits from addressing interoperability issues in human health care, we are interested in exploring if similar benefit and success can be achieved in the space of animal health monitoring; starting by adopting solutions from human healthcare to animal monitoring. The next set of the chapters would present the solution based on the desired abilities that would be required in a farm setup.

Chapter 2 - Animal Telemetry Network

The primary goal of Animal Telemetry Network (ATN) is to achieve a real-time interoperable network of systems that would facilitate: (1) integrating heterogeneous devices and telemetry systems via shared communication infrastructure and (2) hosting applications ("apps") that provide desired monitoring capabilities and scope for automation by acquiring information from and updating/controlling integrated devices.

Animal Telemetry Network (ATN) is composed of components. Components can produce/consume data and trigger/perform actions, a device is typically a component with some associated hardware (e.g., sensor). However, a device may house multiple components depending on how it is constructed. An app is a pure software component. In our demo, we are primarily focused on components. Figure 2.2 shows various components and their interactions. The communication in the network is enabled by a communication substrate.

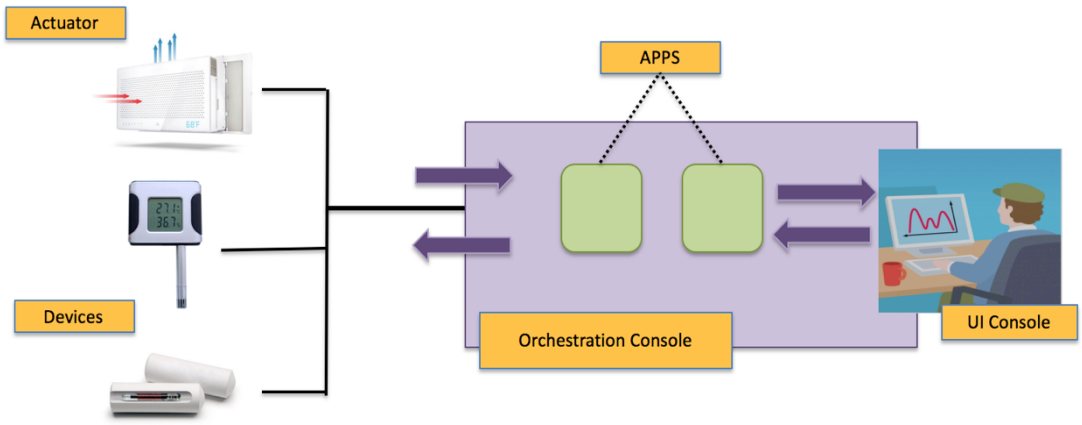

Figure 2.1 Animal Telemetry Network

The app would provide monitoring capabilities to the farm owner. The farm owner would have a library of apps readily available at his/her disposal to choose from based on his requirements. Based on the app functionality, there could be many devices that satisfy the app's

requirement. Hence the farm owner needs to choose the set of devices he wants the app to monitor. We have built the orchestration console for this sole purpose. In the Orchestration console, the farm owner can launch any monitoring app he/she likes from the app library and configure them against a set of devices which he/she intends to monitor.

To be able to compose devices and apps, we need a way to capture their interfaces, capabilities, and functionalities. This is important because we could potentially evaluate the interoperability of two components by just looking at their interface/capability descriptions. The next two chapters will focus on what and how capabilities are captured in our solution and how this enables use to build a more interoperable system.

Chapter 3 - Device Models

The Device Models (DM) aka Device Specification provides a declarative and machine-readable metadata description of the capabilities of the device (e.g., physiological parameters, alerts, communication patterns) exposed over the device's network interface and the DMs are exchanged with the console at association time to form a basis for interoperability [8]. The term device in this chapter is not restricted to only sensors or actuators but it also applies to apps. It can also be termed as component models.

The envisioned device model is somewhat analogous to the IEEE 11073 Domain Information Model (DIM), in that they both provide a declarative description of device capabilities, and they are exchanged with the manager at association time to form the basis of interoperability.

3.1 Device Model Categories

Various device aspects captured in a device model is grouped into four categories. Table 3.1 lists the various device model categories and its description.

DM category	Description
Device Properties	Metadata information about the device Example:- Manufacturer, Device Model, Device type
QOS (Quality of Service)	Information about the guarantees that a network needs to provide for a reliable communication of device data. More about QoS and guarantee in the next chapter. Example:- MinimumSeparation, MaximumLatency
Communication Patterns	Information about the different patterns that are applicable for a device. More about patterns in the next chapter. Example:- Publish, Subscribe etc.
Data Properties	Description about the data handled by the component. Example:- DataType, MinimumValue, MaximumValue, Unit etc.

Table 3.1 Device Model Categories

3.2 Device Model Properties

Table 3.2 captures describes the properties of each category, its description, data type and an example value that were relevant at the time of doing this project.

DM category	DML Property	Description	Data Type	Example Value(s)
Device Property	DeviceName	Name of the device	String	Bolus Sensor
	Manufacturer	Name of the manufacture of the device.	String	Smart Stock
	DeviceModel	Model number of the device being used.	String	SS01
	DeviceType	Type of the device being used.	String	Sensor, Actuator
	OutputName |InputName	Name of the output port or input port.	String	Temperature
Data Properties	Unit	Standard unit used to measure the value captured	String	Celsius
	DataType	Data type of the payload		String.class
	Minimum Value	Least value that the captured value can take	Integer	10
	Maximum Value	Highest Value the captured value could ever possibly reach	Integer	100
Communication Patterns	Publish | Subscribe| Send|Receive| Initiate|Execute	Type of Communication Pattern	n/a	publish
QoS properties	Datasize	Data size of the payload	n/a	4.bytes
	Frequency	Frequency of publication	n/a	10000.milliseconds
	Maximum Latency	Maximum duration to service a (data/action)	n/a	2000.milliseconds

		request.		
	Minimum Separation	Minimum duration between two consecutive communication events	n/a	8000.milliseconds
	Maximum Separation	Maximum duration between two consecutive communication events	n/a	19000.milliseconds

Table 3.2 Device Model Properties

The device model/specification is expressed in a domain-specific language (DSL). We realized this DSL as an internal DSL in Groovy as Groovy is a flexible language and creating a DSL is Groovy was relatively easy; we could have done the same in Ruby or Lisp. Figure 3.1shows a sample device model file of an Ambient Temperature Sensor.

The DM-based description of a device's capabilities is used in multiple ways to provide automatic support for implementation and use of the device interface. During device development, the development environment automatically generates APIs that a device implements to provide the services implied by the DM description. Figure 3.2 illustrates this process. At run time, upon device registration (will be discussed more thoroughly in chapter 5), console is made aware of the device capabilities in order to support communication associated with the physiological parameters, alerts, and device settings specified in the device's DM.

```
package edu.ksu.ccsatn.device.examples

import edu.ksu.ccsatn.device.builder.DeviceBuilder

Integer.metaClass.getBytes = { -> delegate }
Integer.metaClass.getMilliseconds = { -> delegate }

def device = new DeviceBuilder().build{

        deviceName 'Digital Temperature Device'
        manufacturer 'Adafruit'
        deviceModel 'DS18B20'
        deviceType 'Sensor'


        outputs{
                output{
                        outputName 'AmbientTemperature'
                        unit 'celsius'
                        datatype Double.class
                        minimumValue (-55)
                        maximumValue (+125)
                        publish{
                                datasize 4.bytes
                                frequency 10000.milliseconds
                                maximumLatency 2000.milliseconds
                                minimumSeparation 8000.milliseconds
                                maximumSeparation 19000.milliseconds
                        }
                }
        }

}
```

Device Properties

Data Properties

QoS Properties

Figure 3.1 Sample Device Specification file

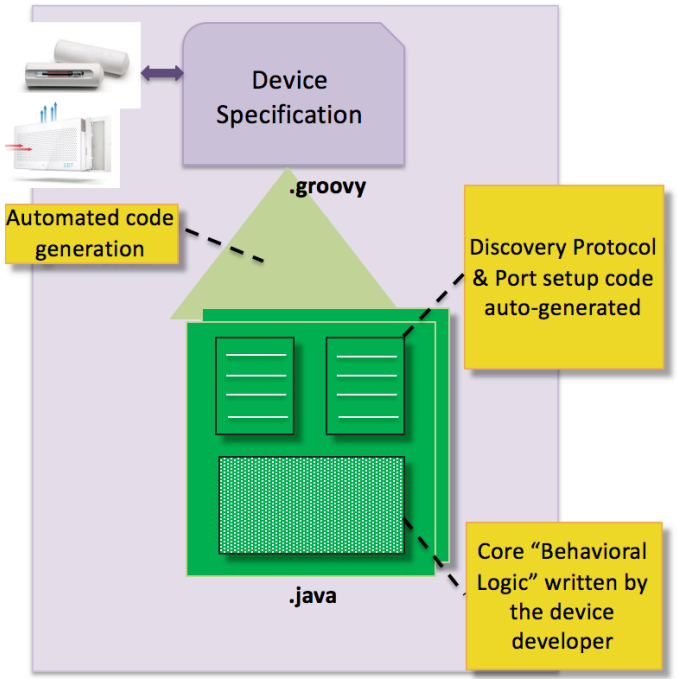

Figure 3.2 Auto Device Interface Code Generation

Each app uses the DM language to specify the device capabilities that it requires to carry out its intended function. This DM-based approach is important for supporting integration function provided by the ATN run-time environment; namely, when a user configures after the

launch of an app, the console will automatically check that the capabilities specified by a device's DM are compatible with the app's requirements. Since both the app's requirements on devices as well as devices' provided services are specified in the DM language, this automated compatibility check is made significantly easier. In addition, during app development, the same DM-to-API code generation used for devices is used to generate APIs that the app employs to access device capabilities. Uniform and automatic generation of APIs on both the "service side" (i.e., the device) as well as the "client side" (i.e., the app) leads to a fundamental property: compatibility between an app and devices on the DM level guarantees interoperability between an app and devices at the code/middleware level. Figure 3.3 describes this process.

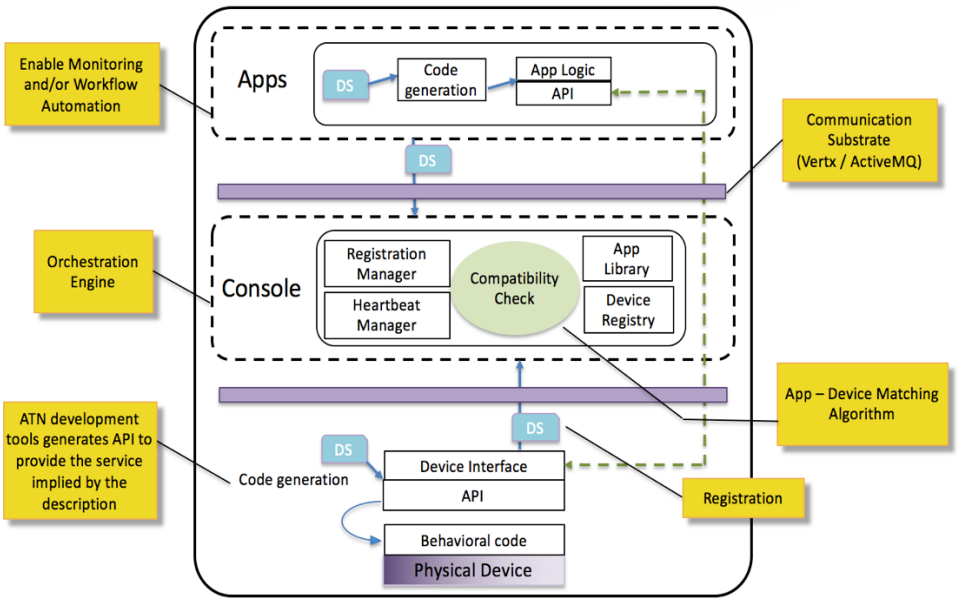

Figure 3.3 Architecture of the ATN

Chapter 4 - Communication Patterns

In this chapter, we describe our adoption of the set of communication patterns [7] proposed for medical domain and how we adapted them for animal telemetry network with minor alterations; specifically, identifying new QoS properties relevant to animal telemetry.

As is, the communication patterns from medical domain abstract away the details of communication tasks, reduce engineering overhead, and ease compositional reasoning of the system. These patterns have been successfully implemented on top of two distinct platforms (i.e., RTI DDS [11]and Vert.x [9]) to allow for experimentation. As part of this effort, we have extended the implementation to operate on top of ActiveMQ [10].

4.1 Failures in ATN

Here we have described a few issues that might arise during the operation of the ATN that could cause failures in the network and result in undesirable outcomes.

- **Congestion -** Local Area Network (LAN) available in the farms usually are very similar to what we find at our homes. The routers used in these networks are capable of handling data flow between 54 Mbit/s to 600 Mbit/s in case of 802.11n router and even the older set of routers that are available such as the 802.11b can handle up to 5.6Mbit/s. These are sufficient numbers to handle data flow in case of telemetry networks. To support this argument, consider a farm with a 1000 telemetric devices deployed on the network. The internal clocks among these devices may/may not be synchronized, the transmission period of these devices may/may not be same. Assume in a worst case scenario we have all device's clock synchronized and have the same transmission period. To calculate the amount of data on the network, lets assume each data packet size to be 100 bytes (32-byte data (Rumen Bolus data packet size) + 60 byte padding (header TCP and other layers)),

so at worst we would have 800 Kbits on the network. These numbers are less compared to what the routers available today are capable of handling. In a case where only a small portion of bandwidth is available for ATN, congestion in the network can be avoided if we knew the data size and the transmission frequency of all the devices. Given these two parameters when adding a new device to the network, the network can determine if adding this device would cause congestion in the network or not and the ranch owner is left to make the decision on adding the device to the network.

- **Fast Publication –** Fast Publication occurs when a device publishes data more often than it is configured to do. For example, say the device was supposed to publish in every 10 minutes but it happened to publish a data at the 5th minute, this is case of fast publication. This is interesting because the device has behaved in a way it isn't supposed to behave i.e. it has deviated from its specification. And since most of the telemetric devices are battery operated, this behavior might reduce the battery life. This may cause issues at subscriber end too if subscriber is capable of consuming a certain number of message per second, then at max it should only be presented with that many. If it is presented with more than what the subscriber can handle, it will drop those messages and the user needs to be notified of such messages.
- **Slow Publication –** Slow publication occurs the device publishes later than it is supposed to publish. This is interesting because if you look at the overall scale of data that we should have received by the end of the hour/day, we are not receiving as much as we should and we might be falling behind and in this case it might not be possible for the farm owner to be able to decide on the state of the animal.

- **Out of Range or Dead Device –** This is a special case of slow publication where the device doesn't transmit for an infinitely long time either because its gone out of range or dead. When the animal walks into areas from where the signal reception is not so good for the signal receivers or when the network is down, the device may not be able to publish data. If the devices are not embedded, one might be interested to notify the ranch owner of this behavior if the device is capable of producing some kind of audio / visual feedback. If they are not capable of providing such feedback or if they embedded, then one might wish to store those messages onto a buffer and transmit them later when they come back into range or when substrate comes online.
- **Slow Consumption –** Slow consumption occurs if the subscribes are not able to process the messages in time. In such cases the user is to be notified of the slow consumption of the subscriber.

4.2 Quality of Service (QoS) properties.

Having described the failures that can occur in the network, we'll look at how some new QoS (Quality of Service) properties can help detect such cases if they occur.

- *Data packet size & frequency* –Having these two parameters while adding a new device into the network, we can calculate how much amount of data would be on the network in the worst case and decide whether addition of this device would congestion or not.
- *Maximum Latency (x)* – If the communication substrate fails to accept a publish request within 'x' time units, then the publication results in timeout, indicating the possibility of out of range or dead device conditions.
- *Minimum Separation (x)* – If *[Current_transmission_time - Last_transmission_time > x],* it is an indication of Fast publication.

- *Maximum Separation (x)* – If *[Current_transmission_time - Last_transmission_time > x]*, it is an indication of Slow publication.
- *Minimum Separation (x)* (Subscriber end)- If messages arrive at intervals less than 'x', it is an indication of fast arrival of messages. This property will help filter all such messages.
- *Maximum Latency (x)* (Subscriber end)- If the subscriber fails to consume a message within 'x' time units, then the message is considered as an unconsumed message and after a fixed number of consecutive unconsumed messages (specified by *ConsumptionTolerance* sub-property), the subscriber is notified of slow consumption.

4.3 Rationale for communication patterns

This section will explain the rationale for including and excluding aspects of communication from the patterns.

- **Data Type:** For a valid connection between two communication endpoints, both need to agree on the types of the data being communicated.
- **Quality of Service (QoS):** Communications in animal telemetry network may require and impose some guarantees, e.g., notify the ranch owner if a device is publishing faster than normal. Such constraints can impact the behavior of the underlying communication substrate and the communicating components. Furthermore, violation of such constraints can lead to undesirable results, e.g., device runs out of battery sooner. Hence, the proposed patterns capture QoS requirements.
- **Local Control:** Not all QoS properties are supported by all communication substrates. To deal with this possibility, the proposed patterns breakdown common QoS properties into finer properties that can be monitored locally (as part of the client or the service) and

from which common QoS requirements for the underlying communication substrate can be derived.

- **Abstraction:** In component-based approach to software, component frameworks abstract away lower-level details of various aspects, e.g., communication, data persistence. Such abstraction helps component developers to focus on the core behavior of components and delegate lower level details to the framework. Moreover, such abstraction can assist with modeling and reasoning of the components and their composites. In a similar spirit, the proposed patterns abstract the lower-level details of communication substrate.

4.4 Description of various Communication Patterns

To describe each communication pattern, we would use a fixed format that captures the *intent* of the pattern, *description* of the pattern, prescribed *use* of the pattern and *QoS properties* supported by the pattern.

4.4.1 Publisher – Subscriber (Producer - consumer) Pattern

Intent: Decouple publishers (producers) and subscribers (consumers) of data by focusing on the topic of interest (and not on the publishers and subscribers).

Description: In this pattern, *publisher* role publishes data about a *topic* and a *subscriber* role subscribes to data about a topic. (This pattern is an incarnation of topic-based communication offered by most publish-subscribe middleware [3].) The topic uniquely identifies the type of the published/subscribed data. The act of publishing data is asynchronous — the publisher does not wait for the communication substrate to deliver the message to subscribers.

Use: Connect data interfaces not associated with parameters (that affect actions).

QoS Properties: Figure 3.1 illustrates the relationship between the supported QoS properties as the pattern is exercised at runtime.

- *MaximumLatency* (*Lpub*) to accept a publish request. If the communication substrate fails to accept a publish request within L_{pub} time units, then the publication results in *timeout* failure.
- *MinimumSeparation* (*Npub*) between two consecutive publications. If the duration between two consecutive publications is less than N_{pub}, then the second publication is dropped with *fast publication* failure.
- *MaximumSeparation* (*Xpub*) between two consecutive publications. If the duration between two consecutive publications is greater than X_{pub}, then the user is notified of slow publication.

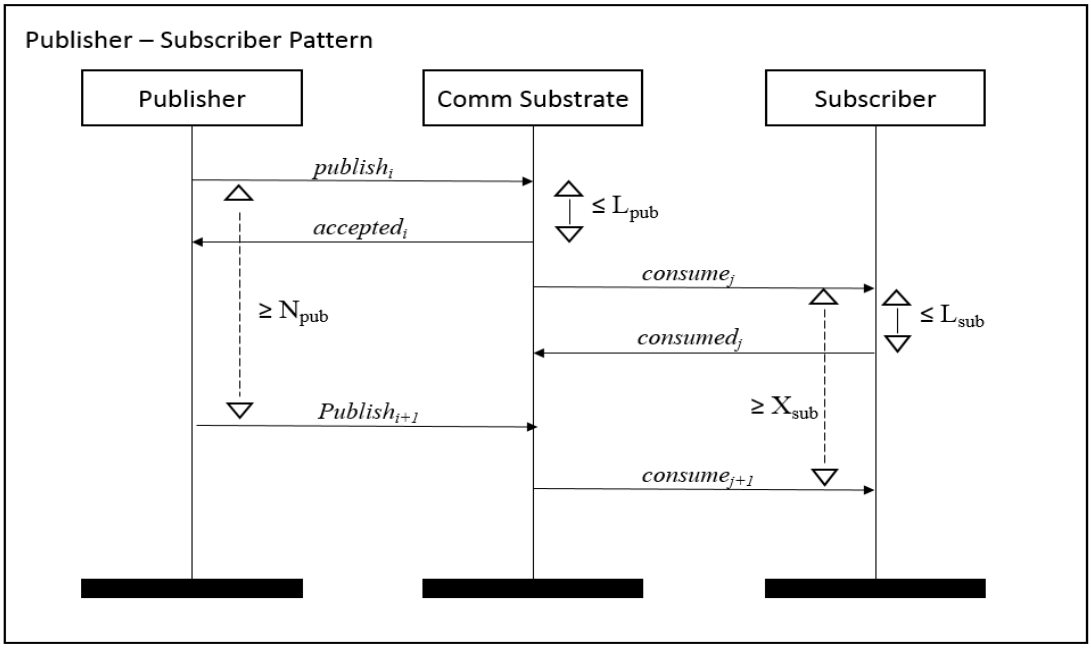

Figure 4.1 Publish-Subscribe Pattern

- *MaximumSeparation* (*Xsub*) between two consecutive message arrivals at the subscriber. If the duration between the arrival of two consecutive messages is greater than X_{sub}, then the subscriber is notified of slow publication.
- *MaximumLatency* (*Lsub*) to consume a message. If the subscriber fails to consume a message within Lsub time units, then the message is considered as an unconsumed message. After a fixed number of consecutive unconsumed messages (specified by *ConsumptionTolerance* sub-property), the subscriber is notified of slow consumption.

4.4.2 Sender – Receiver Pattern

Intent: Provide data to a specific component.

Description: In this pattern, the *sender* role sends data to a specific *receiver* role and the receiver responds back with either *data accepted* or *data rejected* acknowledgement. In terms of data flow, the data travels from the client (sender) to the server (receiver). This pattern requires the sender to know the identity of the receiver. This identity uniquely identifies the sent data and the data type; this enables static validation of the communication. This pattern is synchronous — the sender waits either for an acknowledgement, a notification of failure, or a fixed period, whichever is earlier.

Use: Connect data interfaces associated with parameters.

QoS Properties: Figure 3.1 illustrates the relationship between the supported QoS properties as the pattern is exercised at runtime.

- *MinimumSeparation* (N_{sen}) between consecutive messages sent. If the duration between two consecutive messages sent is less than N_{sen}, then the second request is dropped with *fast send* failure.

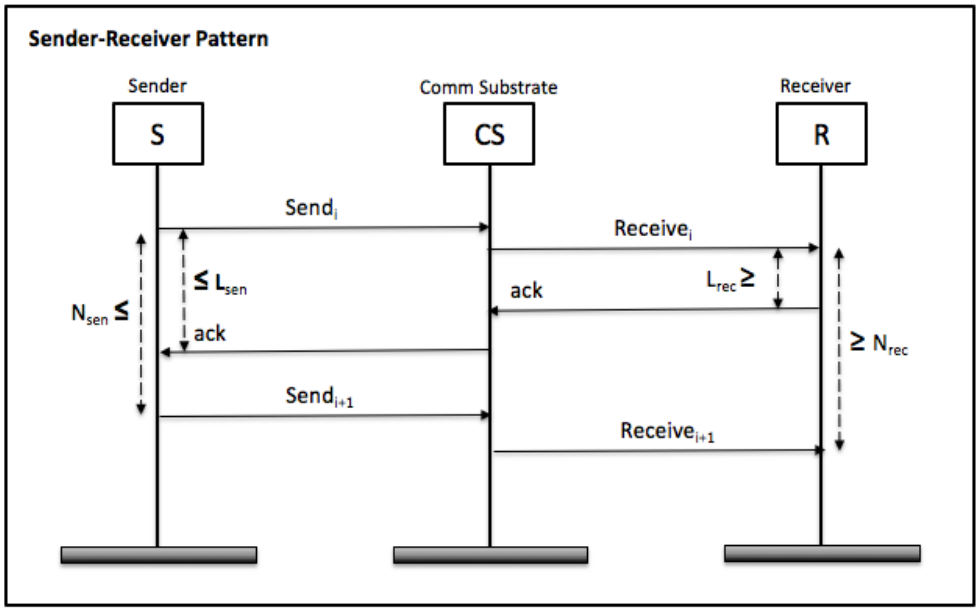

Figure 4.2 Send-Receive Pattern

- *MaximumLatency (*L_{sen}*)* between the sending of a message and the arrival of the acknowledgement. If the response does not arrive within L_{sen} time units, then the request results in *timeout* failure.
- *MinimumSeparation (*N_{rec}*)* between the arrival of messages. If the duration between the arrival of two consecutive messages is less than N_{rec}, then the message is dropped with *excess load* failure.
- *MaximumLatency (*L_{rec}*)* between receiving a message and providing an acknowledgement to the communication substrate. If the acknowledgement is not provided within the L_{rec} time units, the message results in *timeout* failure.

4.4.3 Initiate – Execute Pattern

Intent: Initiate an action in a specific component.

Description: In this pattern, the *initiator* role requests a specific *executor* role to perform an action. Depending on the successful completion of the action, the executor provides *action*

succeeded or *action failed* acknowledgement. If the action is unavailable, then the executor provides *action unavailable* acknowledgement. This pattern requires the initiator to know the identity of the executor. Since the pattern does not facilitate flow of parameters, it is safe to combine this identity with an action identifier provided by the initiator to uniquely identify the action. This pattern is synchronous — the initiator waits either for an acknowledgement, a notification of failure, or a fixed period, whichever is earlier.

Use: Connect action interfaces.

QoS Properties: This pattern supports QoS properties similar to those supported by the sender-receiver pattern. Specifically, the initiator role supports *MinimumSeparation (*N_{ini}*)* and *MaximumLatency (*L_{ini}*)* properties similar to *MinimumSeparation (*N_{sen}*)* and *MaximumLatency (*L_{sen}*)* properties supported by sender role but with *fast initiation* and *timeout* failures, respectively. Similarly, the executor role supports *MinimumSeparation (*N_{exe}*)* and *MaximumLatency (*L_{exe}*)* properties similar to *MinimumSeparation (*N_{rec}*)* and *MaximumLatency (*L_{rec}*)* properties supported by receiver role with the same kinds of failures.

Chapter 5 - Deployment Protocol

Having described about the different components in the network, the device model, the communication patterns, this chapter would focus on the one click deployment of the device and app components.

5.1 About the Protocol

Figure 5.1,5.2&5.3 below are sequence diagrams highlighting the components (Sensors, Actuators and App) involved along with their interactions with one another with respect to time. All of these components can be four different individual entities on a network (i.e. each having their own IP address), but for demo purpose we would be running the app and the console on the same machine. This would be the same machine the farm owner would use to monitor the devices.

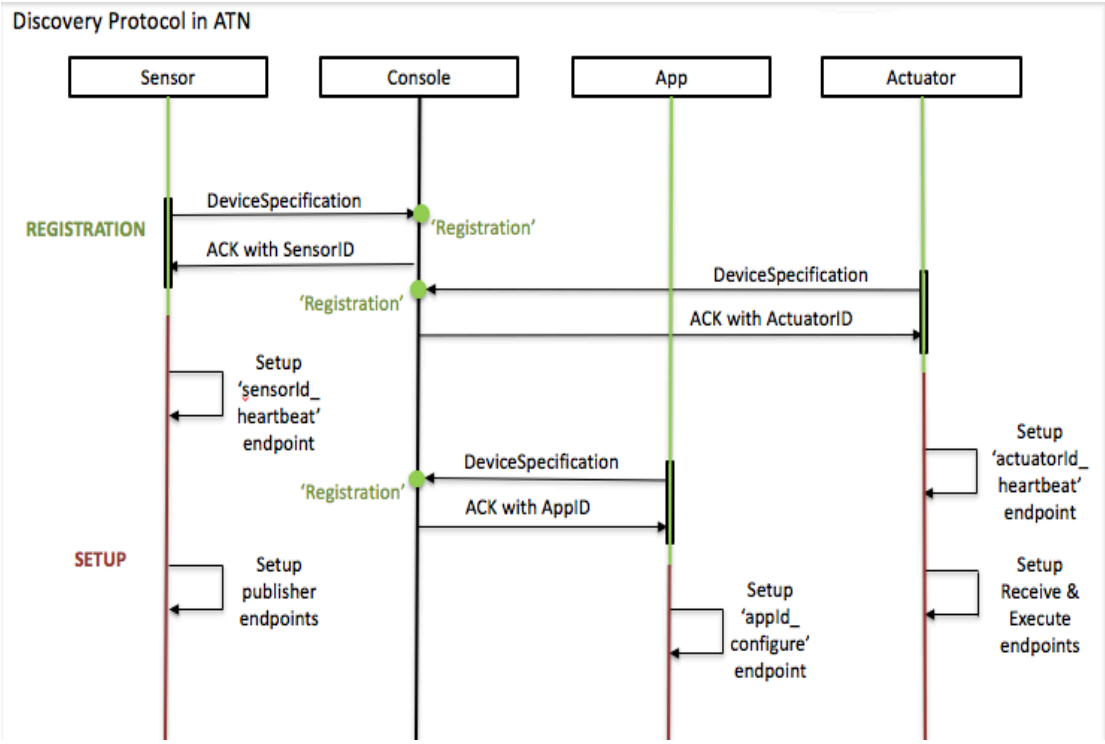

Figure 5.1 Registration and Setup Stage

The protocol consists of three stages, Registration, Setup and Running (or Configured) stage and this would be common across the sensor, app and actuator components. The app component would have an extra configure stage, more details about it is explained below. The protocol would be executed step by step after device startup.

- **Registration** – This is the first step for all components trying to join the ATN for the first time. This step is highlighted on the sequence diagram in green in figure 5.1. During registration, the device or the app would send a registration message (device specification of the component) to an 'registration' endpoint on the console. The console on reception of the device specification would do three things; (i) generate a unique Id for the new device (ii) add a new entry into the 'Device Registry' in case of sensor or actuator or 'App Registry' in case of an app and (iii) send an acknowledgement for the message received along with the unique Id it generated. The registration step is a success if the device or the app successfully receives the Id from the console and moves on to the next step in the process. If not, the device would try to retransmit the registration message until it successfully registers. There is no particular order that is enforced on the registration step, any device or app can register in any order.
- **Setup** – An app or a device would enter into setup stage only if the registration step was successful. The setup stage is highlighted in red in figure 5.2. During this step, the device setup all the endpoints as described in the device specification. e.g. 'AmbientTemperature' endpoint from figure 3.1. The devices in particular would also setup a heartbeat endpoint. This endpoint is used to check if the devices are within range and if they are alive. On the other hand, since more than one connected device can satisfy the device requirements of an app, it needs an input from the user about which device to

connect to and hence it would set up a ‘configure’ endpoint to which configuration information would be sent later in the configure step.

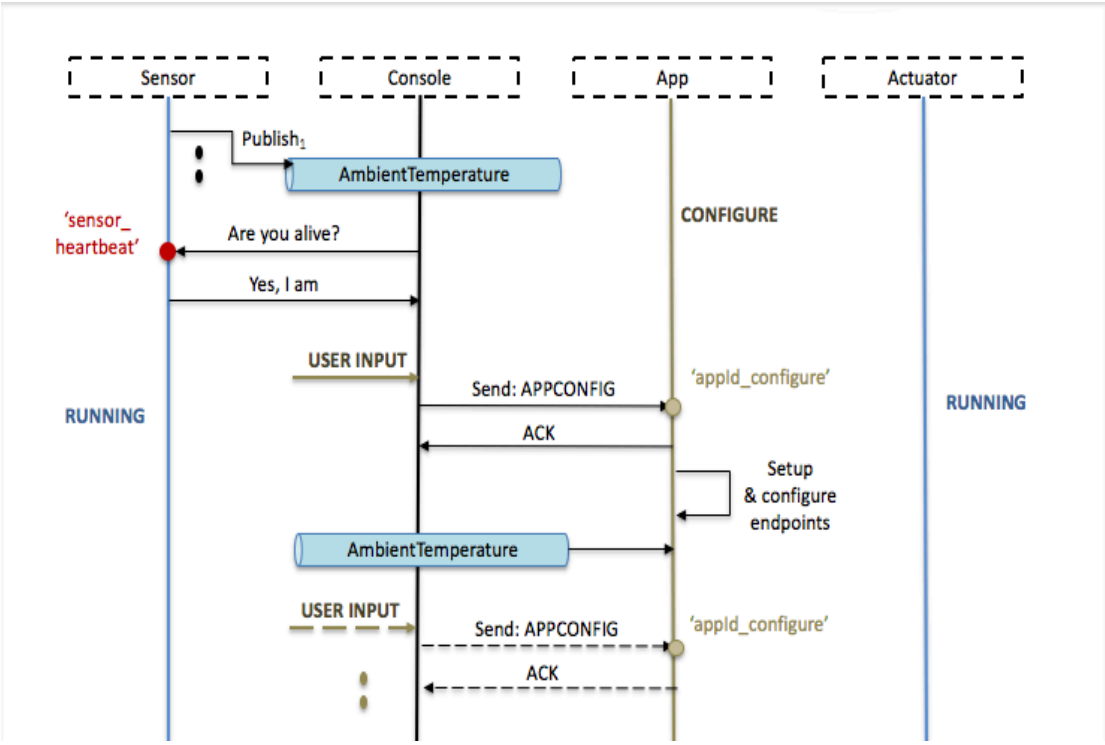

Figure 5.2 Configure Stage

- **Configure** – This step is only applicable to the apps and this process is highlighted in tan on the sequence diagram in figure 5.2. When more than one connected device’s satisfies the device requirements of an app, the farm owner will have to select the device to be connected to the app via the console. Once the farm owner selects his desired configuration, the console conveys the configuration to the ‘configure’ end point that was setup on the app in the previous stage. If at a later point in time, the farm owner decides to change the device that satisfied the device requirement of the app, he can do so by selecting a new device from the matched list of devices and the console will convey the new configuration information to the app. On receiving the configuration information, the app would use this information to set up the app endpoints accordingly.

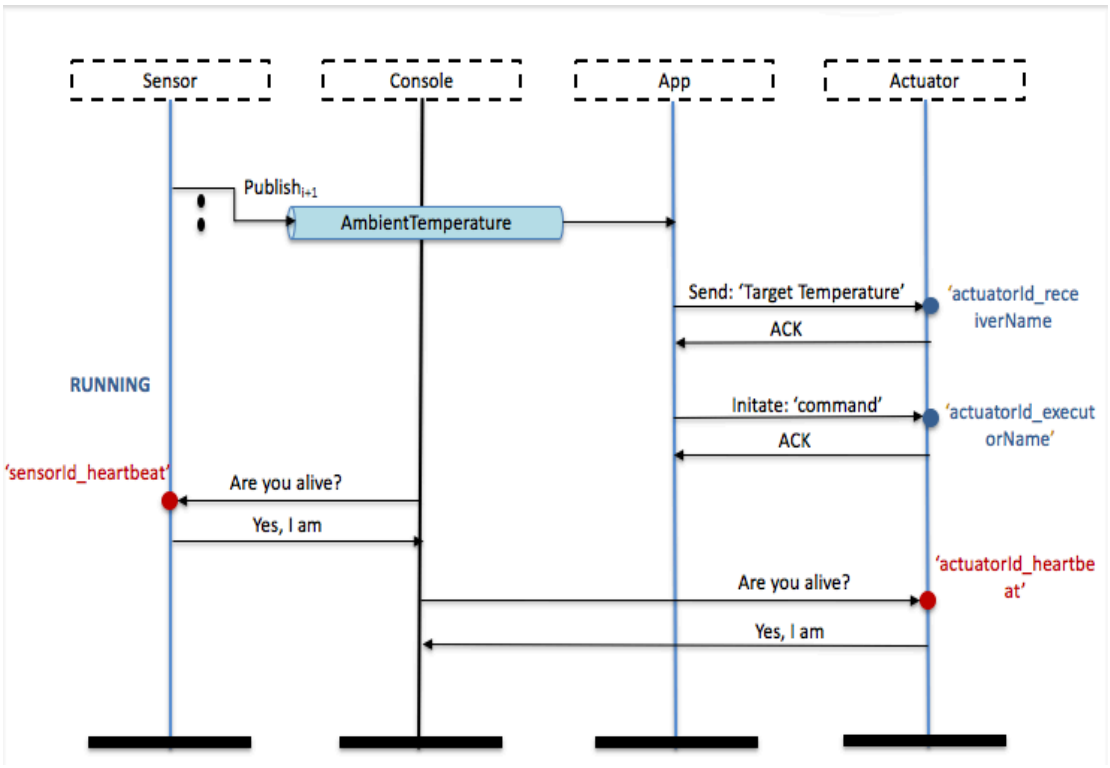

Figure 5.3 Configured System

- **Running** – This stage is highlighted in blue in the sequence diagram in figure 5.3. By this stage, the sensor would be publishing data on a particular topic. The sensor at any point in time can be of out of range and come back right into this same stage. In this stage, the endpoints of the actuator have been configured and the actuator would be waiting to receive commands. The user can check the status of the sensor or the actuator by requesting for a heartbeat through the console. The app in this stage can either be fully or partially configured. The components would continue to exhibit their functionality until they are interrupted or shutdown by the user.

The protocol is triggered when a component is turned on. Every component in the network would follow the above mentioned protocol in order to gain access to the network and be part of the system of systems. The deployment procedure is simplified and setup time is significantly reduced due to automation, the operators of these components or the farm owners need not worry about the setup procedures anymore.

Chapter 6 – Evaluation

Having described a way to achieve interoperability among heterogeneous systems, in this chapter we would evaluate the capabilities of the system. We would also evaluate actions and responsibilities of the device manufacturers, app developers, and farm owners. The evaluation would be done by exercising each step as part of a demo described below.

6.1 Demo Scenario

Imagine a farm with a shed housing a herd of cattle. All of them are embedded with rumen bolus which measures the core body temperature. A temperature sensor placed in the shed to measure the ambient temperature of the shed. An air cooler is used to maintain the temperature inside the shed at desired levels. The farm owner monitors the rumen and ambient temperature and initiates cooling through an app when necessary. A console is used to connect and orchestrate the apps and devices. Figure 2.1 gives a high level view of the components involved in the demo.

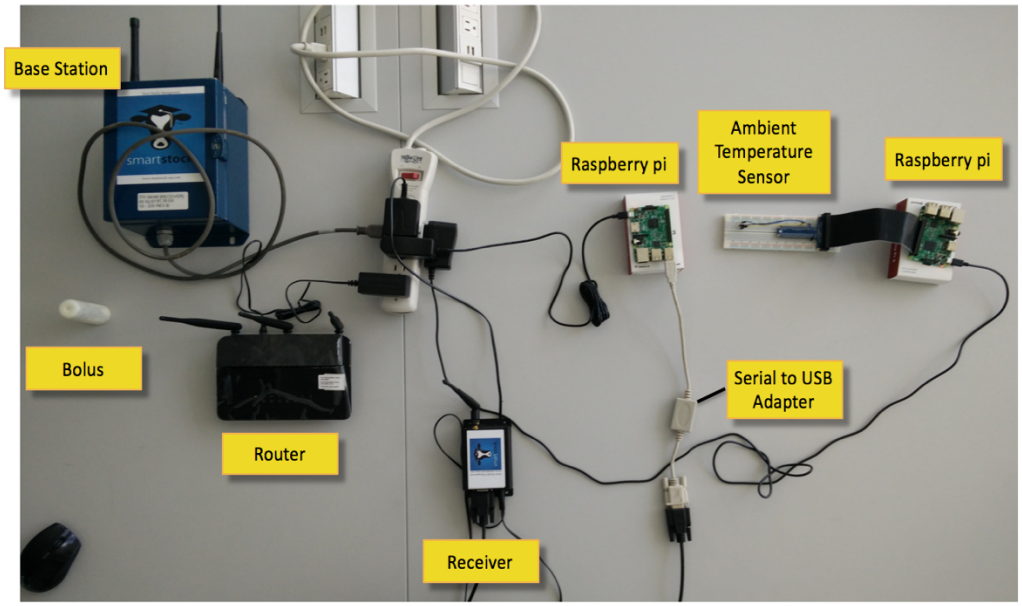

Figure 6.1 Physical Components

Figure 6.1 shows the various physical components. The air cooler in this demo would be represented as a pseudo component. The Raspberry Pi's are used as proxy to connect the bolus and temperature sensor to the network.

6.2 Process Evaluation

The first step in the demo process is for the device manufacturers or the app developers to model their devices. Figure 3.1 serves as an example showing how an Ambient Temperature sensor can be modeled. Once the device capabilities are captured, the device interfaces along with other necessary registration code and protocol would be auto generated. The device developers need to add the behavioral logic to the generated code. Figure 6.1 shows a sample piece of code generated.

```
public String registration() {
    this.filereader = new FileReader();
    this.registrationport = new RegistrationSenderPort(communicationManager);
    String message = filereader.getFile("file/Digital_Temperature_Device.groovy");
    registrationport.setup();

    SendPayload _payload = registrationport.send(message);
    this.deviceID = _payload.getFromReceiverMessage();

    LOGGER.info("RECEIVED DEVICE ID : " + deviceID);
    return deviceID;
}

public void setHB() {

    this.heartbeat = new HeartBeatReplyPort(
            new ResponderConfiguration<>(this.deviceID + "_heartbeat", 1000, 5000, new HeartBeat()),
            communicationManager);
    heartbeat.setup();
}

public void setAmbientTemperaturePort() {
    this.ambienttemperatureport = new AmbientTemperaturePort(new PublisherConfiguration("AmbientTemperature",
            10000, 2000, 8000, 19000, new AmbientTemperaturePort()), communicationManager);
    ambienttemperatureport.setup();
}

public void publishAmbientTemperaturePort(Double data) {
    ambienttemperatureport.publish(data);
}
```

Figure 6.2 Sample code generated for Ambient Temperature Sensor

A desired communication substrate would be chosen which would enable communication across the network. We have support for two different substrates Vert.x[9] and ActiveMQ[10].

A web-based orchestration console is deployed first, the console has a user interface which enables the user to keep track of devices, deploy apps and configure them. Figure 6.3 shows a successfully deployed orchestration console.

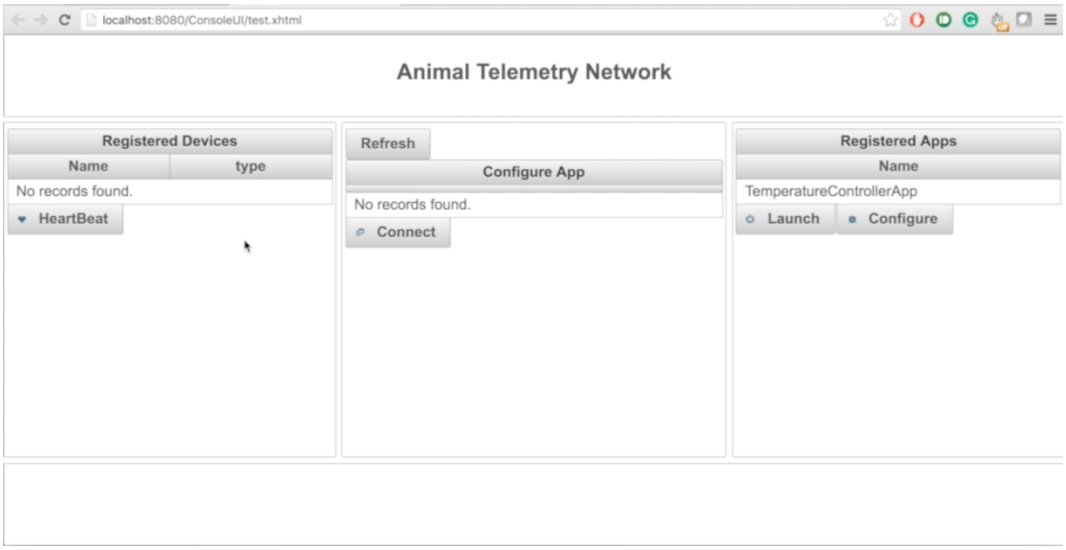

Figure 6.3 Orchestration console

The next step involves deploying devices and apps and configuring the apps against devices to achieve the desired monitoring functionality. Figure 6.4 show the registered devices on the left, a configuration panel in the center showing the matched device ports against the app ports and the apps available on the right.

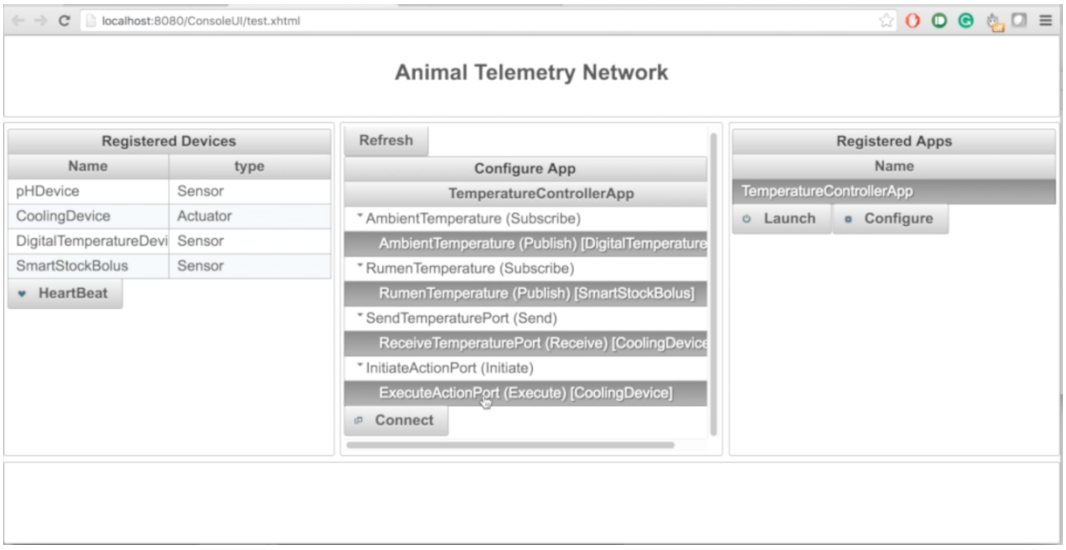

Figure 6.4 Device – App Configuration

If the matching and the connection was successful, then monitoring app should be able to receive information from the ambient temperature device and the rumen bolus and be able to set and initiate the air cooler. Figure 6.5 serves as a proof for the above statement.

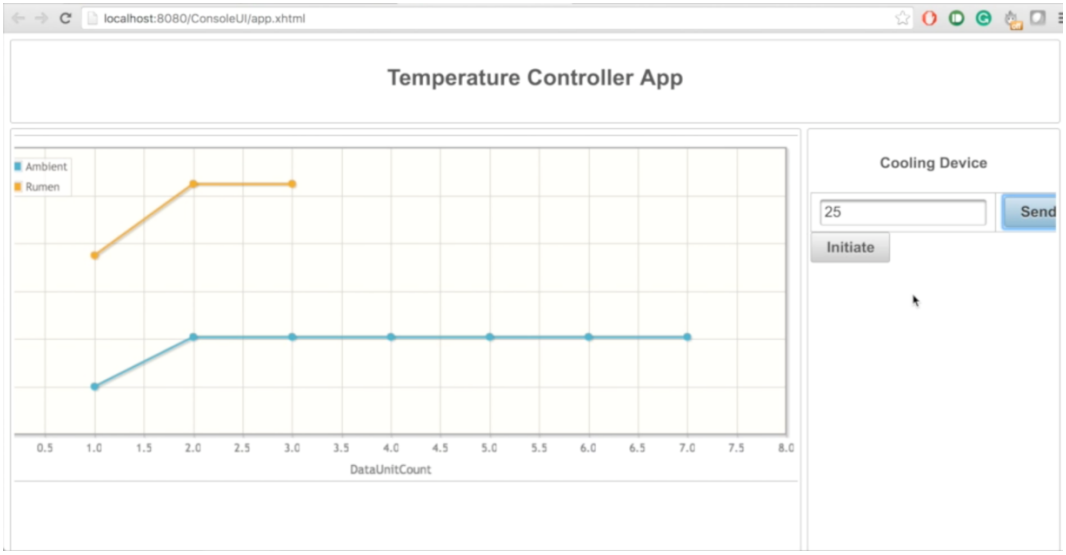

Figure 6.5 Temperature Monitoring App

Chapter 7 – Contribution

Although we borrowed quite a few concepts and design choices from existing solutions which addressed the heterogeneous and interoperability issues in the field of medical science, the manner in which they were implemented in ATN were different. This section will describe some of the designs and implementation which were new or different from the existing solution.

The goal of device models was to capture the device capabilities along with device interface specification. The models also needed to be readable, extensible and simple enough for device manufacturer to easily write it. In order to meet these requirements, we chose Domain Specific Language (DSL). Many programming languages provide DSL support. We used the DSL support provided by Groovy. The reason being Groovy is built on Java and since all our implementation was in Java, it was easier to translate DSL into Java.

Communication in any domain can be categorized into a set of standard communication patterns. We looked at the common communication patterns that occur in the field of animal telemetry and have categorized them accordingly. A minimum set of Quality of Service (QoS) properties were proposed for the animal telemetry that guarantees reliability and provides notification against any component deviating from their specifications.

To address the deployment issues with the current telemetry setup, we designed a deployment protocol that enabled a one click deployment procedure for any component in the system. Lastly, we provided a basic console (user interface) to be used by the farm owners that help him keep track of his devices along with a configuration panel to configure the apps. We also provided tabs for each app for desired monitoring capability.

Chapter 8 - Summary

We began by presenting a sketch of issues with the current setup in the farms, and we set out to see if we could address those issues and present a solution. We proposed ATN which was step closer to achieving interoperability in the field of animal health monitoring. We addressed the interoperability issue by exposing the device's capabilities over the network using Device Models (DM). This enabled the network to be aware of its devices and was easier to achieve interoperability. The apps delivered the required monitoring capabilities around heterogeneous devices and has paved a path to deliver workflow automation. Communication in the network was standardized with a set of communication patterns that captured properties that guaranteed reliable network and guard against achieving undesirable outcomes. At last the long deployment and setup time issues was addressed by the discovery protocol that automated the setup procedure.

For more information about the work described in this report, please reach out to me via http://bitbucket.org/ashwinkrishna or Santos Research Laboratory at contact@santoslab.org.

References

[1] Internet of Things Global Standards Initiative [Online]
Available: http://www.itu.int/en/ITU-T/gsi/iot/Pages/default.aspx [Accessed: July 6, 2016]

[2] K. Ashton That "Internet of Things" thing RFiD Journal (2009)

[3] H. Sundmaeker, P. Guillemin, P. Friess, S. Woelfflé, Vision and challenges for realizing the Internet of Things, Cluster of European Research Projects on the Internet of Things—CERP IoT, 2010.

[4] J. Hatcliff, A. King, I. Lee, A. MacDonald, A. Fernando, M. Robkin, E. Vasserman, S. Weininger, and J. M. Goldman. Rationale and architecture principles for medical application platforms. ICCPS, 2012.

[5] Andrew King, Sam Procter, Daniel Andresen, John Hatcliff, Steve Warren, William Spees, Raoul Jetley, Paul Jones, and Sandy Weininger. An open test bed for medical device integration and coordination. In Proceedings of the 31st International Conference on Software Engineering (ICSE), pages 141–151, 2009.

[6] Venkatesh P Ranganath. MDD4MS: Model Driven Development for Medical Systems. December 3, 2014.

[7] V. P. Ranganath, Y. J. Kim, J. Hatcliff, and Robby, "Communication patterns for interconnecting and composing medical systems," in Engineering in Medicine and Biology Society (EMBC), 2015 37th Annual International Conference of the IEEE, Aug 2015, pp. 1711–1716.

[8] Y. J. Kim, J. Hatcliff, Robby and Weininger, Sandy. "Integrated Clinical Environment Device Model: Stakeholders and High Level Requirements."

[9] Vert.x [Online] Available: http://vertx.io/ [Accessed: August 1, 2016]

[10] ActiveMQ [Online] Available: http://activemq.apache.org/ [Accessed: August 1, 2016]

[11] RTI Connext, 2015. Available at http://www.rti.com/ products/dds/index.html.

Appendix: Code Snippets

Appendix includes the ambient temperature sensor's specification file along with the generated Java code from the specification file.

DSL Snippet

```
package edu.ksu.ccsatn.device.examples

import edu.ksu.ccsatn.device.builder.DeviceBuilder

Integer.metaClass.getBytes = { -> delegate }
Integer.metaClass.getMilliseconds = { -> delegate }

def device = new DeviceBuilder().build{

	deviceName 'Digital Temperature Device'
	manufacturer 'Adafruit'
	deviceModel 'DS18B20'
	deviceType 'Sensor'


	outputs{
		output{
			outputName 'AmbientTemperature'
			unit 'celsius'
			datatype Double.class
			minimumValue (-55)
			maximumValue (+125)
			publish{
				datasize 4.bytes
				frequency 10000.milliseconds
				maximumLatency 2000.milliseconds
				minimumSeparation 8000.milliseconds
				maximumSeparation 19000.milliseconds
			}
		}
	}

}
```

Device Properties

Data Properties

QoS Properties

Generated Java Code for the above DSL

```
package device;

import java.io.BufferedReader;

@SuppressWarnings("all")
public class DigitalTemperatureDevice {
    private static final Logger LOGGER = LoggerFactory.getLogger(DigitalTemperatureDevice.class);

    private RegistrationSenderPort registrationport;
    private HeartBeatReplyPort heartbeat;
    private AmbientTemperaturePort ambienttemperaturereport;
    private CommunicationManager communicationManager;
    private String deviceID;
    private FileReader filereader;

    public DigitalTemperatureDevice() {
        communicationManager = new CommunicationManagerImpl(0, "127.0.0.1");
        communicationManager.setUp();
    }

    public String registration() {
        this.filereader = new FileReader();
        this.registrationport = new RegistrationSenderPort(communicationManager);
        String message = filereader.getFile("file/Digital_Temperature_Device.groovy");
        registrationport.setup();

        SendPayload _payload = registrationport.send(message);
        this.deviceID = _payload.getFromReceiverMessage();

        LOGGER.info("RECEIVED DEVICE ID : " + deviceID);
        return deviceID;
    }

    public void setHB() {

        this.heartbeat = new HeartBeatReplyPort(
                new ResponderConfiguration<>(this.deviceID + "_heartbeat", 1000, 5000, new HeartBeat()),
                communicationManager);
        heartbeat.setup();
    }
    public void setAmbientTemperaturePort() {
        this.ambienttemperaturereport = new AmbientTemperaturePort(new PublisherConfiguration("AmbientTemperature", 4,
                10000, 2000, 8000, 19000, new AmbientTemperaturePort()), communicationManager);
        ambienttemperaturereport.setup();
    }

    public void publishAmbientTemperaturePort(Double data) {
        ambienttemperaturereport.publish(data);
    }

    public static void main(String[] args) {
        DigitalTemperatureDevice diSensor = new DigitalTemperatureDevice();
        String deviceId = diSensor.registration();
        if (deviceId == null || deviceId == null) {
            LOGGER.info("Exception: DeviceID received during registartion is NULL");
            new Exception("Error during registartion!");
        } else {
            diSensor.setHB();
            diSensor.setAmbientTemperaturePort();
        }

        try {
            Process p = Runtime.getRuntime().exec("./ambient_tmpr.py");
            BufferedReader stdInput = new BufferedReader(new InputStreamReader(p.getInputStream()));
            BufferedReader stdError = new BufferedReader(new InputStreamReader(p.getErrorStream()));
            String s = null;
            while ((s = stdInput.readLine()) != null) {
                diSensor.publishAmbientTemperaturePort(Double.parseDouble(s));
            }
        } catch (Exception e) {
            System.out.println("exception happened - here's what I know: ");
            e.printStackTrace();
        }
    }
```

```java
package device.ports;

import java.lang.Double;

@SuppressWarnings("all")
public class AmbientTemperaturePort extends PublisherCallback<Double> {
    private static final Logger LOGGER = LoggerFactory.getLogger(AmbientTemperaturePort.class);

    private PublisherConfiguration publisherConfiguration;
    private CommunicationManager communicationManager;
    private Publisher publisher;

    public AmbientTemperaturePort() {
    }

    public AmbientTemperaturePort(PublisherConfiguration publisherConfiguration,
            CommunicationManager communicationManager) {
        this.publisherConfiguration = publisherConfiguration;
        this.communicationManager = communicationManager;
    }

    public void setup() {
        Pair pair = this.communicationManager.createPublisher(this.publisherConfiguration);
        this.publisher = (scp.api.Publisher) pair.second;
        if (CommunicationManager.Status.SUCCESS == (scp.api.CommunicationManager.Status) pair.first) {
            LOGGER.info("publisher AmbientTemperature created");
        } else {
            LOGGER.error("Error during publisher AmbientTemperature creation");
        }
    }
    public void publish(Double data) {
        LOGGER.info("Publishing....." + data);
        this.publisher.publish(data);
    }

    @Override
    public void handleTimeOutMessage(Double message) {
        LOGGER.warn("Message timed out! Handling...");

    }

    @Override
    public void handleSlowPublication() {
        LOGGER.warn("Slow Publication Warning!Handling...");

    }

    @Override
    public void handleFastPublication() {
        LOGGER.warn("Slow Publication Warning!Handling...");

    }
}
```

BNF grammar for writing Device Specification

<DeviceSpecification>	::== <DeviceProperties> <Ports>
<DeviceProperties>	::== <DeviceName> <Manufacturer> <DeviceModel> <DeviceType>
<Ports>	::== “Outputs” <Port> | “Inputs” <Port>
<Port>	::== “Output” <DataProperties> <CommPattern>
	| “Input” <DataProperties><CommPattern>
<DataProperties>	::== (<OutputName> | <InputName>) <Unit> <DataType> <MinimumValue> <MaximumValue>
<CommPattern>	::== <Pattern> <QoSProperties>
<Pattern>	::== “Publish” | “Subscribe” | “Initiate” | “Execute” | “Send” | “Receive”
<QoSProperties>	::== <datasize> <frequency> <MaximumLatency> <MinimumSeparation> <MaximumSeparation>