

Using REDS to Implement a Publish-Subscribe Middleware for Online Fleet Management and Precision Agriculture

Thomas Jakobsen¹⁾, Lars M. Kristensen²⁾, Raphael Dobers²⁾, Claus Aage Grøn Sørensen³⁾, Iver Thysen³⁾

¹⁾ Alexandra Institute Ltd., Aabogade 34, DK-8200 Aarhus N, Denmark, thomas.jakobsen@alexandra.dk

²⁾ University of Aarhus, Department of Computer Science, Aabogade 34, DK-8200 Aarhus N, Denmark, {kris, dobers}@daimi.au.dk

³⁾ University of Aarhus, Inst. for Jordbrugsproduktion og Miljø, Blichers Allé 20, DK-8830 Tjele, Denmark, {claus.soerensen, iver.thysen}@agrsci.dk

This research is funded by the Nordunet3 program, <http://www.nordunet3.org>.

Abstract

It is an ongoing trend that more and more information technology is incorporated into modern farms in order to increase productivity and comply with various documentation provisions. However, the potential of this is often not fully utilised due to the heterogeneous nature of the underlying communication technologies. Often, applications are unable to communicate because they reside on networks with incompatible protocols. This is especially true for applications involving mobile equipment such as tractors, harvesters, handheld devices, and small computational devices attached to animals. In this paper, we propose a solution to this problem and validate it with software prototypes based on the REDS framework. Our solution involves a middleware layer implemented in software that hides the differences in the various underlying networks. Applications running on top of the middleware are thereby allowed to communicate in a uniform manner. Since some of the underlying networks will be mobile ad-hoc networks, the uniform abstraction for communication is based on the publish-subscribe paradigm rather than the well-known client-server paradigm.

1 Introduction

Recent developments in agricultural technology have seen ample information technology capabilities being offered to farmers. A modern farm typically consists of a number of mobile entities equipped with computers and various sensors. Technologies such as WiFi, GPRS, Bluetooth, and ZigBee are essentially able to connect these to computers on the farmers' Local Area Network (LAN), as well as PDAs, mobile phones, and even servers on the Internet. Together, this scenario forms a complex and heterogeneous network infrastructure.

The capabilities inherent in these technologies are, however, often not fully utilized because of difficulties in transferring data from machine to machine or between machines and the Farm Management Information System (FMIS). A comprehensive, efficient, and robust data communication between these entities is essential in order to establish the basis for intelligent real-time monitoring, operations optimisation, and documentation [1,3]. This is especially the case for a successful deployment of innovative technologies like semi-autonomous or autonomous vehicles in agriculture, where issues such as automated scheduling, routing, real-time monitoring of vehicles and materials are essential [2].

The principal objective for mobile communicating applications is thus to be able to continuously collect and distribute data and information among work units, users, sites, and applications as the basis for dynamic resource allocation and scheduling. The data must be filtered and delivered to potential users at the right time. The current technology being used for on-line communication between mobile work units and the FMIS is most often the traditional client-server architecture using, for example, mobile telephone communication [4,5]. However, this paradigm has its drawbacks in the form of message congestion, lack of fault-tolerance, lack of adaptability to shifting conditions, and carrier costs. In contrast, a combination with a mobile ad-hoc network [6], based on e.g. the publish-subscribe paradigm [7], is expected to exhibit robustness, flexibility, and scalability in terms of dynamically adapting to changing configurations of mobile work [8]. In this way, the publish-subscribe paradigm addresses many of the challenges inherent in emerging mobile applications within the agricultural domain.

The rest of this paper is organised as follows. Section 2 presents the reference network architecture that we have developed and introduces the basic ideas of mobile ad-hoc networks. Section 3 discusses the publish-subscribe paradigm and explains how it is supported by the REDS framework. Section 4 defines three representative case studies from the agricultural domain: local situation awareness, data registration for documentation as well as traceability purposes, and centralised fleet management. This section also illustrates how applications for the cases are easily implemented using the REDS system. Finally, Section 5 concludes and discusses future work.

2 Network Architecture and Ad-hoc Networks

Our approach is based on certain assumptions about the typical network infrastructure found in a modern farm. We expect that not all parts of current farms are completely covered by a fixed infrastructure such as local area networks (LANs) and wireless access points. Efficient communication in these areas is instead implemented by means of mobile ad-hoc networks.

2.1 Reference Network Architecture

The network infrastructure envisioned is outlined in Figure 1. Here, a number of mobile entities are forming one or more mobile ad-hoc networks based on a wireless technology, e.g. WiFi [9]. These entities could be tractors and harvesters working on a field, animals with attached computing devices, or even small robots controlling weeds or performing other autonomous tasks in the field. If a mobile entity is out of radio range to other entities or the farm LAN wireless router, and maintaining connectivity to other entities such as a central data collection server is critical, GPRS or UMTS wireless technologies could be employed as a backup connection or simultaneously on selected nodes.

We also assume that servers of various kinds are located on a nearby LAN, typically in a farm building. In addition, more servers may exist elsewhere on the Internet, e.g. databases containing information about crops and pests. Finally, we assume that handheld devices such as mobile phones and PDAs are utilising different kinds of wireless communication based on e.g. GPRS or UMTS, or short-range technologies such as Bluetooth and ZigBee if they have no WiFi-based mobile ad-hoc capabilities. Some of the mobile entities may be connected to routers (gateways) that route data traffic between the ad-hoc networks and the LAN. Yet other routers are responsible for routing between the LAN and the Internet.

We argue that today, a farm containing all these different network facilities has a large wasted capability due to the difficulties that applications will experience when they have to communicate across the heterogeneous set of underlying networks. As an example, it is currently far from simple for a stand-alone application on the farm LAN to seamlessly communicate with a set of tractors moving around in a part of the nearby field. It is not trivial, either, for a small application running on a farmer's cell phone to communicate with sensors placed in the field or in a nearby stable.

As mentioned in the introduction, this paper proposes a solution to this problem that involves a middleware layer of software aimed at hiding the details related to communication across heterogeneous networks and routing in ad-hoc networks. The middleware layer instead presents a set of simple and powerful communication primitives on which robust domain specific applications can easily be built.

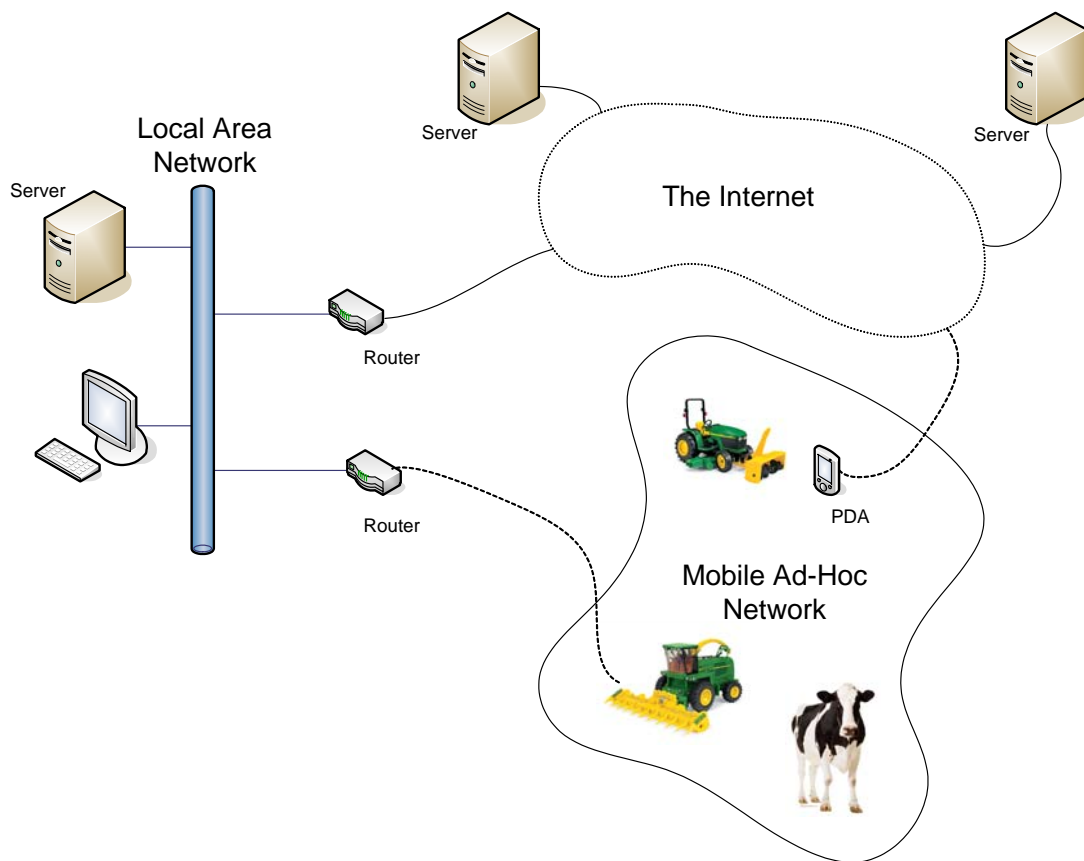


Figure 1: Reference network architecture.

2.2 Mobile Ad-Hoc Networks

A *mobile ad-hoc network* is a wireless network in which the nodes move around arbitrarily and where each node is willing to forward data for other nodes in the network. Hence, a node can send data to another node even though the two nodes are not within immediate range of each other. Mobile ad-hoc networks should be distinguished from managed wireless networks where all nodes must be within range of a dedicated node called an *access point* that is the only node which forwards data. Figure 2 illustrates the difference between mobile ad-hoc and managed wireless networks.

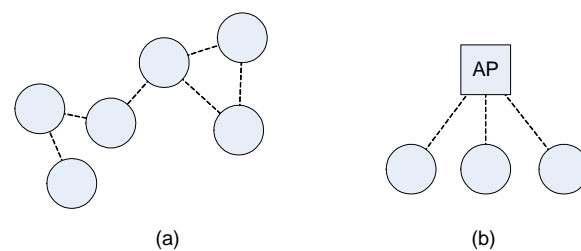


Figure 2: (a) Mobile ad-hoc network. (b) Managed wireless network.

The fact that mobile ad-hoc networks do not require any central nodes to forward messages makes them suitable in some environments where managed networks cannot operate. This is likely to be even more evident due to the fact that the size of computing devices is continuously decreasing and that they are being built into more and more everyday devices. In fact, support for mobile ad-hoc networks is already built into industry standards such as ZigBee [10] and to some extent WiFi [9].

The process of forwarding data in a network is referred to as *routing*. Routing in an ad-hoc network is a difficult task due to the mobility of the nodes, and developing efficient and robust protocols for routing in such networks is an active research area. Several protocols have already been developed [11]. These protocols are generally divided into *proactive* and *reactive* protocols.

In a proactive routing protocol, individual nodes continuously maintain some information about the current position of other nodes in the network. This information is typically stored in tables stating, "If I receive a message designated for this node, then forward the message to that node". In reactive protocols, such routing information is instead computed on demand, typically the first time a message designated for a particular node has to be forwarded or

when a message has to be forwarded and the current forwarding route is no longer valid due to changes in the network topology.

Proactive protocols are able to route a series of messages to a receiver with minimal delay since each node already knows where to forward a message when it arrives. However, there is a performance penalty for continuously keeping all the routing information up to date when the network topology changes, and proactive protocols are therefore more likely to cause message congestion. In reactive protocols, on the other hand, a message sender is exposed to a longer delay when starting to send messages to a receiver, but the overall amount of data sent between nodes in the network is typically considerably less.

3 Publish-Subscribe Systems and REDS

In order to present the domain specific applications with a simple and consistent set of primitives for communication, the design of the middleware layer mentioned in the introduction has to reflect a uniform communication paradigm. This Section introduces the publish-subscribe paradigm, compares it to the well-known client-server paradigm, and introduces the REDS framework, which is an implementation of the publish-subscribe paradigm for use in mobile ad-hoc networks or normal wired networks.

3.1 The Publish-Subscribe Paradigm

In the publish-subscribe communication paradigm [7,12], senders do not send messages to specific receivers. Instead, messages are divided into classes and a sender publishing a message belonging to a certain class has no knowledge of whom – if anyone – receives that message. Receivers, on the other hand, subscribe to one or more of the classes and subsequently receive only the messages published that belong to these classes. A receiver does not know who actually sent a message being received. One node in the network can have both the role of sender and receiver, sometimes even of the same class of messages. In terms of the publish-subscribe model, senders are called *publishers* and receivers are called *subscribers*. Typically, a subscriber is only interested in a subset of all the published messages. Therefore, a publish-subscribe system must implement a filtering mechanism, which ensures that only the subscribed messages are delivered to a certain node. This can generally be achieved in two ways, called *topic-based* filtering and *content-based* filtering [7].

In a topic-based system, messages are published to named logical channels called *topics*. Subscribers then subscribe to one or more of these topics and subsequently receive all

messages published to these topics. Hence in topic-based systems it is the publishers that are defining the classes of messages. In content-based systems, however, a subscriber does not subscribe to a certain topic. Instead, the subscriber defines some constraints on the contents of messages and subsequently receives only the messages with a content that matches these constraints. That is, in content-based systems, the subscriber defines the classes of messages. In both cases a network of dedicated nodes called *brokers* is responsible for routing and filtering the published messages to the subscribing receivers. In implementations of publish-subscribe systems, the broker role may be independent of the publish/subscribe role in the sense that some or all of the publishers or subscribers may be brokers, but brokers can also be nodes that do not themselves publish or subscribe to classes of messages.

One advantage regarding performance is that most publish-subscribe systems rely on some broadcast primitive which fits naturally with the underlying hardware broadcast facilities of current technologies such as WiFi [9] and Ethernet (IEEE 802.3). The main advantage, though, of the publish-subscribe paradigm compared to the well-known client-server approach is the loose coupling between senders and receivers of messages: Applications running on a publish-subscribe based system are not required to know the existence of other senders and receivers and can therefore continue to function normally whether or not specific senders or receivers are present. In the traditional tightly coupled client-server paradigm, a client needs to know about the existence of a specific server and vice versa. That is, a client cannot send messages to a server unless the client knows the network address of the server and the server is currently running. Also, the server cannot send messages back if the client is not running.

Apart from the decoupling of locations of publishers and subscribers, a publish-subscribe system also allows a temporal decoupling: If a message is sent on a particular topic and a certain subscriber to that topic is not currently present, the broker system can place the message in a buffer until the subscriber returns. Thus, contrary to the client-server paradigm, publishers and subscribers do not need to consider whether a message was published when subscribers were present or not. The message is guaranteed to arrive at the subscribers as long as there is enough buffer capacity in the broker system. Finally, since the broker system in some publish-subscribe systems is distributed and replicated between several independent nodes in the network, there is a much higher degree of fault-tolerance

compared to relying on one or more centralised servers available only through a single point of access. Hence, one part of the network can continue to be operational despite losing connectivity to another one, and transparently synchronise information upon reconnection between the two.

Many applications that are currently implemented using the client-server paradigm contain logic that attempts to take care of some of the communication issues mentioned above. For applications that are running on top of mobile ad-hoc networks, where the topology is continuously changing and presence of specific nodes at a certain time is never really guaranteed, we argue that such applications are better implemented based on a publish-subscribe system where these communication issues are instead handled efficiently and uniformly by the underlying publish-subscribe system.

3.2 REDS: A Reconfigurable Dispatching System

REDS (REconfigurable Dispatching System) [13,14] is a Java implementation of a publish-subscribe system. We have chosen this system as basis for our prototype due to several attractive features. First, REDS is seemingly the only publish-subscribe system that is explicitly designed to support arbitrary topological changes in the underlying network. That is, previous systems, while focusing on scalability, tend to impose some kind of limitation on the mobility of the nodes in the network. Furthermore, the REDS system has a very modular architecture. For instance this means that the system is designed in such a way that the routing of messages that is characteristic for publish-subscribe systems is decoupled from the underlying mechanisms that connect the network. Furthermore, the specific strategies for routing and subscription are encapsulated in modules that can relatively easily be changed without affecting the rest of the system. This, combined with the availability of the source code, allows middleware programmers to easily adapt the system to their own special needs. For instance, it is easy to define custom formats for messages and filters, custom mechanisms used internally by each broker to match and forward messages, as well as custom strategies for routing messages. Finally, REDS natively supports replies to messages: Brokers have the ability to store the route of a message allowing subscribers to send a reply directly back to the publisher. This kind of built-in bidirectional communication loosens up the strict decoupling between publishers and subscribers described in Section 3.1 and thereby enables the system to handle some applications that do not naturally fit the pure publish-subscribe paradigm. An example of such an application is described in Section

4.2 and the issue of coupling between publishers and subscribers is further discussed in Section 5.

3.2.1 The REDS Software Architecture

The Application Programmers Interface (API) offered by REDS to the developer is simple and reflects the fact that REDS is independent of the specific implementation of message and subscription formats. An abstract class `Message` and an interface `Filter` encapsulate these details. The system is accessed via an instance of `DispatchingService` on which the application programmer simply invokes the methods `publish()` and `subscribe()`. When subscribing, an instance of the `Filter` class is given as parameter and subsequently, only messages matching the given filter are received. As examples, it could be text messages with filters based on regular expressions or XML messages with filters based on XPath.

The interface `ComparableFilter` that extends `Filter` encapsulates functionality for determining whether a filter covers another filter, i.e. matches at least the same messages. This information is used to better control the propagation of subscriptions in the broker network, thereby allowing messages to be routed more efficiently as described by Carzaniga et al [15].

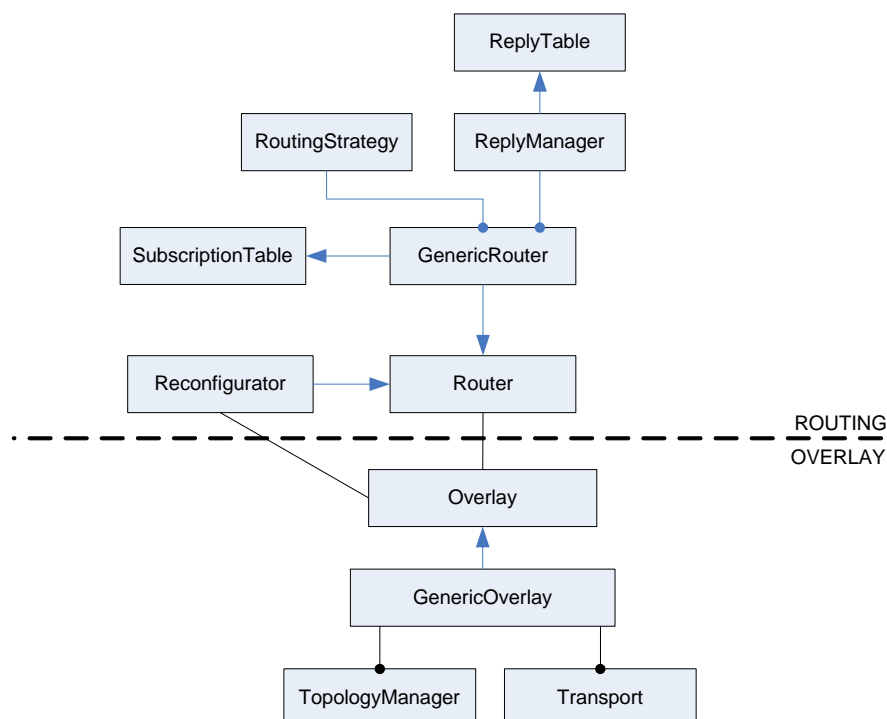


Figure 3: The internal structure of a REDS broker.

The architecture of the system is best understood by looking at the internal structure of a REDS broker node shown in Figure 3. The system is generally divided into two layers. The overlay layer is responsible for maintaining the broker network despite dynamic reconfigurations of the network nodes. The broker network is an *overlay* network since it does not necessarily correspond to the underlying physical network. On top of the overlay layer, the routing layer is responsible for the routing of published messages. The system is event-based, meaning that in a specific broker node, the routing component connects to the overlay component and subsequently receives notifications when subscriptions, messages, and replies arrive from neighbour brokers.

The overlay layer is divided into a `Transport` component and a `TopologyManager` component. The `Transport` component is responsible for communication between two broker nodes that are adjacent in the overlay network and it is implemented either via TCP or UDP. The `TopologyManager` maintains the overlay network despite dynamic reconfigurations of the network. The system comes with two concrete implementations, an `LSTreeTopologyManager` that is well suited for routing in wired networks, and – more interestingly in our case – a `WirelessTopologyManager`, that handles dynamic reconfigurations in a mobile ad-hoc network. The `WirelessTopologyManager` basically provides an implementation of a protocol developed by Cugola et al [13]. This protocol is designed especially for routing in a mobile ad-hoc network in a way that enables a publish-subscribe system with content-based filtering to be built on top of it. It is a modification of the MAODV protocol [16] designed for multicast communication in mobile ad-hoc networks, and MAODV is again a modification of the AODV protocol [17], which is a basic reactive routing protocol for mobile ad-hoc networks.

At the routing layer, a `Router` interface implemented by a default `GenericRouter` class is responsible for routing messages. This responsibility is delegated to three sub-components, namely

- A `SubscriptionTable` in which subscriptions are stored. This component is responsible for matching incoming messages against the stored filters to figure out where to forward the message. REDS includes a generic and relatively inefficient implementation based solely on the `match()` method required by the `Filter`

interface, but it is possible to implement more efficient strategies, e.g., by imposing some restrictions on the structure of message and subscription formats.

- A `RoutingStrategy` that implements subscription message forwarding as described by Carzaniga et al [15].
- A `ReplyManager`, which manages replies to messages. The default implementation saves information in a `ReplyTable` when messages arrive and sends back replies immediately, but other strategies are also possible.

Whereas the `Router` assumes a stable network, the `Reconfigurator` is responsible for updating the message subscription system when reconfigurations of the broker network occur. The technique for doing this and the interplay between `TopologyManager` and `Reconfigurator` is described further by Cugola et al [18]. It should be noted that routing takes place at two different levels in the system: The various instances of `TopologyManager` each contain logic for routing messages from one node to another in the overlay network. On the other hand, the components at the routing layer in Figure 3 assume an already connected overlay network and are instead concerned with the content-based routing of messages in a publish-subscribe context, which includes the maintenance of subscription tables.

4 Case Studies

This Section describes three basic cases that we consider to be highly relevant in farms with a network infrastructure similar to the one outlined in Figure 1. We show how solutions to these are implemented using the REDS framework. The cases presented are supposed to illustrate the diversity of applications that are suitable for a publish-subscribe system like REDS. By listing concrete samples of code, we emphasise how relatively straightforward the implementations become when using the high-level primitives of a publish-subscribe system. The communication aspects of the implementation are typically reduced to publishing and subscribing to a couple of channels. The case studies are:

- Local situation awareness exemplified by the information flow needed to coordinate a number of work units collaborating in the execution of a given task.

- On-line and off-line data registration for documentation and traceability purposes. Messages can e.g. be continuously published by mobile entities, stating current GPS coordinates and the currently applied level of fertilisation or spraying. This enables central registration of these data for reporting to authorities.
- Centralised fleet management. Subscriptions of information from multiple farms can support a centrally placed dispatching manager in the task of coordinating a fleet of vehicles operating on multiple farms.

4.1 Case 1: Local Situation Awareness

Our first case belongs to the class of applications that supports local situation awareness. In its simplest form, this could be achieved by letting each mobile entity equipped with a GPS receiver publish its current position. All other interested entities then subscribe to the class of all published GPS coordinates and display these, possibly combined with identifying names or colours, on a screen. In a large farm or contractor setting, we assume that even this simple case will help to improve the overview and coordination of work.

One could imagine several extensions to this case. Some machines might only be interested in knowing the location of other equipment on the field at which the machine itself is currently situated, or a harvester might only be interested in the current location of a certain type of tractor with a transport unit. Other applications in this category combine the immediate situation awareness described above with various kinds of automated reasoning. This could involve automatic notifications published by a nearly filled up harvesting machine. The automatic notifications can trigger sending an empty transport unit to the proper location before the other runs completely full, thereby avoiding costly delays. A concrete implementation of such an application (which is not based on publish-subscribe middleware) is described by L. Aa. Jensen and C. B. Madsen [19].

4.1.1 Implementation

In the following code example, we restrict ourselves to the simplest useful setup, i.e. where GPS coordinates are published by the mobile entities that are equipped with GPS receivers and where each mobile entity subscribes to all published coordinates. In order to implement a solution for this case, we will first have to assume some reasonable interface to the GPS

device. For our illustrating purposes, the interface in Listing 1 will do. A call to `receive()` returns a text representation of the latest received GPS coordinate or blocks the calling thread temporarily until a coordinate becomes available.

```
interface GpsReceiver {  
  
    /**  
     * Gives queue access to received GPS coordinates. A call to this method  
     * removes a coordinate from the queue and returns its text  
     * representation. If the queue is empty, the calling thread is blocked  
     * until a new GPS coordinate is received.  
     */  
    String receive() throws InterruptedException;  
  
}
```

Listing 1: An interface to a GPS receiver.

The next piece of code, the class `GpsPublisher` in Listing 2, is the program supposed to be run by those hosts that are equipped with GPS units. The `main()` method first launches a new thread in which a new instance of a class, `ManetBroker`, is instantiated. This class contains the REDS implementation of a broker in a mobile ad-hoc network. In order for the system to work properly, one broker must be running on all the mobile nodes forming the ad-hoc network.

Then a `PublisherThread` is started. The purpose of this thread is to instantiate a `DispatchingService` object, which basically contains the API to the REDS system. Once the `DispatchingService` has been created, our thread goes into a loop where each GPS message received from the GPS device gets published. Note that we here simply invoke `publish()` on the dispatcher without first having to define a specific channel (or topic). This is due to the fact that REDS uses content-based routing where the subscriber rather than the publisher defines the channels, cf. Section 3.

```

public class GpsPublisher {

    // ports that various components of REDS use for communication
    private static final int brokerPort = 9000;
    private static final int topMgrPort = 9001;
    private static final int localPort = 9002;

    public static void main(String[] args) {

        // start the local broker
        new Thread(new Runnable() {
            @Override
            public void run() {
                new ManetBroker(Transport.UDP, brokerPort, topMgrPort);
            }
        }).start();

        // start the publisher
        new Thread(new PublisherThread()).start();
    }

    private static class PublisherThread implements Runnable {

        @Override
        public void run() {

            // start the dispatching service
            DispatchingService service = new UDPDispatchingService(
                "127.0.0.1", brokerPort, localPort);

            try {
                service.open();
            } catch (ConnectException e) {
                e.printStackTrace();
                return;
            }

            // launch the GPS receiver
            GpsReceiver gps = new GpsReceiverImpl();

            while (true) {
                try {
                    // prepend 'gps:' to received coordinate and publish
                    TextMessage msg = new TextMessage("gps:"
                        + gps.receive());
                    service.publish(msg);
                } catch (InterruptedException e) {
                    return;
                }
            }
        }
    }
}

```

Listing 2: A class for publishing GPS coordinates.

The next code sample, the `GpsSubscriber` in Listing 3, is the program that subscribes and displays the GPS coordinates. It can be run independently of the publish program above. First, a broker thread is launched the same way as in `GpsPublisher` above. Then the program subscribes using a `TextFilter` matching all messages containing the word “gps”. In this example, received messages are then simply printed out on the screen. In a real setting, positions would rather have been graphically displayed on a map. It is worth noticing that `TextFilter` is only an example of a filter that defines the class of messages that the program wishes to receive. Any other filter could have been used.

Figure 4 is a screenshot from a simple demo based essentially on the code listed above. In this demo, three laptops each equipped with WiFi and GPS receivers are publishing their coordinates. They also subscribe to the class of all coordinates which are displayed in a simple graphical user interface. Due to the REDS ad-hoc routing features described earlier, the GUI is able to show all locations despite the fact that the laptops move around and the direct WiFi connection between two of the laptops sometimes breaks.

```

public class GpsSubscriber {

    // ports that various components of REDS use for communication
    private static final int brokerPort = 9000;
    private static final int topMgrPort = 8001;
    private static final int localPort = 8002;

    public static void main(String[] args) {

        // start the local broker
        new Thread(new Runnable() {
            @Override
            public void run() {
                new ManetBroker(Transport.UDP, brokerPort, topMgrPort);
            }
        }).start();

        // start the subscriber
        new Thread(new SubscriberThread()).start();

    }

    private static class SubscriberThread implements Runnable {

        @Override
        public void run() {

            // start the dispatching service
            DispatchingService service = new UDPDispatchingService(
                "127.0.0.1", brokerPort, localPort);

            try {
                // Opening dispatching service
                service.open();
            } catch (ConnectException e) {
                e.printStackTrace();
                return;
            }

            // subscribe to messages containing the word 'gps'
            TextFilter flt = new TextFilter("gps", TextFilter.CONTAINS);
            service.subscribe(flt);

            while (true) {
                TextMessage msg = (TextMessage) service.getNextMessage();
                service.reply(new TextMessage("reply to: " +
                    msg.getData(), msg.getID()));
                System.out.println("Received GPS: " + msg);
            }

        }

    }

}

```

Listing 3: The GpsSubscriber class.

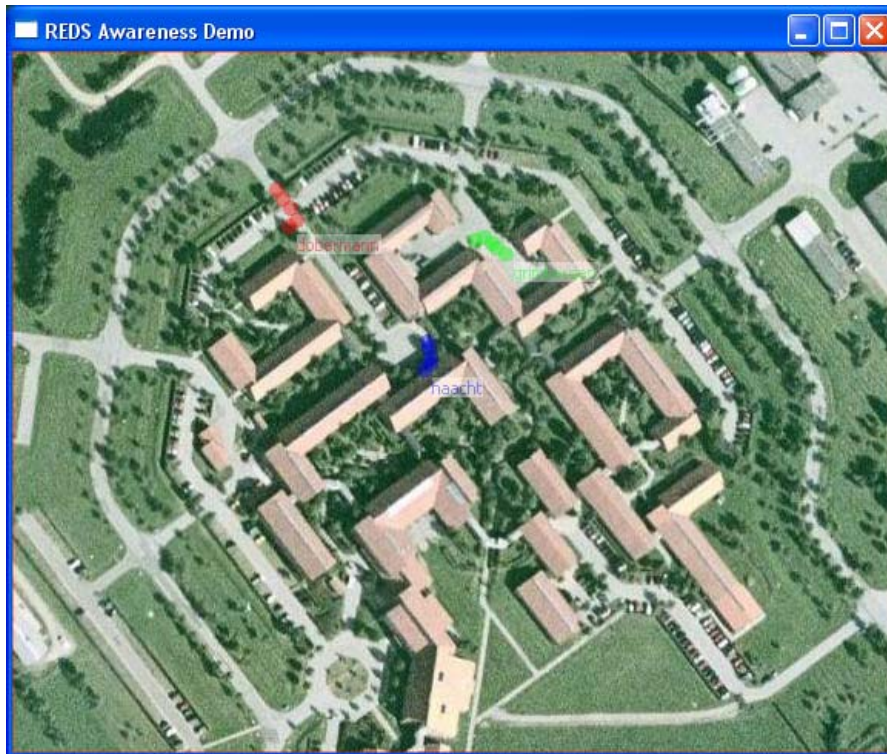


Figure 4: Screenshot from case 1 demo.

4.2 Case 2: Data Registration

This case covers the applications where various kinds of data are published by farm vehicles, animals, or other entities on the farm. Examples of data could be current time, GPS coordinates, current amount of fertiliser in a spraying vehicle, power consumption, or health measures for specific animals. What basically distinguishes this case from the previous case is that the subscriber is now a central database, either on the local farm, or somewhere on the Internet. The data also has a more persistent value, e.g. it could be data used for traceability or documentation issues such as logging, or statistical analysis.

Notice, that even complex applications of this kind, where many types of data from various entities have to be collected, combined and recorded, possibly even in multiple databases at different locations, can relatively easily be implemented on top of a publish-subscribe system like REDS that provides content-based filtering. In case of critical error logging and related applications, we mentioned earlier in Section 3.2 that REDS provides a reply mechanism that allows for a somewhat tighter coupling between publisher and subscriber which is certainly desirable in such situations. The code snippets in Listing 4 and Listing 5 illustrate this.

Suppose that the last part of the publish program from the previous case (Listing 2) instead contains the code in Listing 4.

```
// publish an error message
TextMessage msg =
    new TextMessage("IMPORTANT: This harvester is broken!");
MessageID id = msg.getID();
s.publish(msg);

// receive reply
try {
    Message reply = s.getNextReply(id);
} catch (TimeoutException e) {
    System.out.println("No replies received");
}
```

Listing 4: Code for an error log publisher.

```
// subscribe to messages starting with the word 'IMPORTANT'
TextFilter flt = new TextFilter("IMPORTANT", TextFilter.BEGINS);
s.subscribe(flt);

while (true) {
    TextMessage msg = (TextMessage) s.getNextMessage();
    s.reply(new TextMessage("this is a reply to " +
        msg.getData(), msg.getID()));
    System.out.println("Logging message: " + msg);
}
```

Listing 5: Code for an error log subscriber.

Here, instead of a continuous stream of GPS positions, a single message is published, stating that the publishing harvester is not functioning properly. Then a reply is requested, and if no reply is received within a certain time limit, an exception is thrown. The corresponding subscriber program is outlined in Listing 5.

This subscriber could be a central error-logging program located at the farm centre. By sending a direct reply back, the publisher obtains a guarantee that the messages it has sent have actually been received by the error logger and are not just silently discarded by the publish-subscribe system when no central logger is registered.

4.3 Case 3: Centralized Fleet Management

Our final case is concerned with centralised fleet management. Assume a machine contractor that operates a fleet of expensive equipment such as harvesters and tractors. Farmers when needed, whereby the farmers save the cost of purchasing their own machines, hire individual machines from the fleet. When the fleet reaches a certain size, optimal management and coordination of the fleet, especially during periods of peak activity, will have a significant impact on the operational efficiency and the profitability of the enterprise.

Given a layer of middleware as the one described in this paper, we argue that it will be relatively easy to develop applications supporting dispatching and coordination of such a central fleet, compared to the situation where the applications must handle communication across multiple networks themselves. As an example, consider a manager located at a central facility subscribing to orders and dispatching work units to specific locations. The information made readily available to the manager by the publish-subscribe system would enable quick and dynamic changes and re-allocations of fleet units. This case illustrates the situation where utilisation of the publish-subscribe system goes a step further than coordination on a single farm and perhaps connection to a single database on the Internet. Here, the publishing and subscription of messages span several farms and a central fleet management location.

The future clearly dictates increased integration with management functions in terms of turning fleet management into dedicated planning tools [20]. It includes real-time management comprising looking at current fleet locations and where it should be tomorrow integrating how the fleet behaved today. By increasing the analytical power of such systems it will be possible to analyse, in real-time, the cost implications of accepting or rejecting specific dispatch orders. Improved planning tools combined with ICT systems for monitoring and documentation may increase the capacity utilisation and subsequently improve the timeliness of the field operations. Studies have shown that the capacity utilisation can be increased significantly and thereby reduce the unit costs by e.g. 20-30% by improving the planning and monitoring of mobile units [21]. Experiences from non-agricultural businesses (companies with operating vehicles or service-oriented companies with fleet objects) show significant benefits, e.g. 30% less administrative costs and 80% improvement in detection of malfunctions or poor operational performance because of close monitoring [22]. We argue

that our layer of publish-subscribe middleware could potentially cause similar operational benefits by enabling information flow and integration from different domains.

5 Conclusion and Future Work

In this paper we have presented a publish-subscribe based middleware system for the agricultural domain based upon the Reconfigurable Dispatching System (REDS). This publish-subscribe system supports multi-hop communication between mobile entities that are not within direct range of each other and is suited for the mobile ad-hoc networks that are part of our reference network architecture. The applicability of this approach has been validated by demonstrating how it can be used to implement three representative use cases from the agricultural domain: local situation awareness, on-line data registration, and fleet management. In particular, we have argued that the publish-subscribe paradigm in many cases is a better choice for applications in mobile ad-hoc networks than the usual client-server paradigm.

It should be mentioned, however, that some kinds of applications do not fit naturally to the publish-subscribe paradigm. This involves applications that need an even stronger delivery guarantee than that given by the best-effort strategy available via the buffer mechanism in the publish-subscribe system: Consider a publisher sending messages to subscribers in a publish-subscribe system. If some subscribers are not currently present, the broker system will usually keep the messages in some kind of buffer (possibly distributed among the broker nodes) until the buffer is full and then start to discard messages. This is a best effort strategy, but the publisher may desire the even stronger delivery guarantee that either the message is delivered to all interested subscribers or otherwise, messages are returned stating that some interested subscribers did not receive the message. Implementing this kind of strong delivery guarantee on top of a publish-subscribe system is possible, but tends to violate the whole idea behind the publish-subscribe paradigm, namely that publishers and subscribers should not need to care about the specific senders and receivers of messages. Using the client-server paradigm, it is comparatively straightforward to obtain the guarantee by letting each receiver send back an acknowledgement message when a message is received.

Another case, where publish-subscribe systems may not be appropriate, arises when publishers implicitly make assumptions that subscribers are listening. Consider as an example of this, a farm that uses a publish-subscribe system where various applications

publish log messages to a certain “error log” topic as in the use case 2 example in Section 4.2. If any application senses an error, it publishes a warning under the error log topic. An application in the main building of the farm subscribes to this topic and stores all error messages in a central database. But if the subscriber crashes, the publishers have no way of knowing this (without violating the central idea of publish-subscribe) and will continue to publish error messages that will now simply vanish from the system. Here, again, a tight coupling between sender and receiver is a desired feature. The case study in Section 4.2 illustrates how REDS to a certain extent remedies this by introducing the reply mechanism, allowing publishers and subscribers to be more tightly coupled without introducing overhead in the application layer.

Currently, the REDS system is not designed to handle multiple IP subnets and therefore cannot directly be used for all applications in our case studies. Future work includes adapting the system to such settings. Here we want to develop a system that supports several REDS “islands” to be connected. We are currently investigating whether this can be achieved using the JXTA framework [23]. The JXTA framework provides a set of protocols that enable geographically distributed and networked entities to be interconnected via a peer-to-peer overlay network. Extending the publish-subscribe system to have global coverage will significantly expand the class of agricultural applications that can be handled with our approach as many of these rely on databases residing on Internet servers.

6 References

- [1] F. Kuhlman and C. Brodersen. Information Technology and Farm Management: Developments and Perspectives. *Computers and Electronics in Agriculture*, 30:71-83, 2001.
- [2] C. G. Sørensen, T. Bak, and R. N. Jørgensen. Mission Planner for Agricultural Robotics. AgEng, 2004.
- [3] H. Auernhammer. Precision Farming – the Environmental Challenge. *Computers and Electronics in Agriculture*, 10:31-43, 2001.
- [4] F. Mazzeto, S. Landonio, and M. Vaccaroni. Farm Activity Information System based on Automatic Detection of Machinery Use. *Precision Agriculture*, 1997.
- [5] L. Aa. Jensen, C. G. Sørensen, and R. N. Jørgensen. Real-time Internet-based Traceability Unit for Mobile Payload Vehicles. In *XXXII CIOSTA-CIGR Section V Conference: Advances in Labour and Machinery Management for a Profitable Agriculture and Forestry*, pages 368-374, 2007.
- [6] Charles E. Perkins. *Ad Hoc Networking, An Introduction*, pages 1-28. Addison-Wesley Longman Publishing Co., Inc., 2001.
- [7] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The Many Faces of Publish/Subscribe. *ACM Comput. Surv.*, 35(2):114-131, 2003.
- [8] G. Cugola and H. A. Jacobsen. Using Publish/Subscribe Middleware for Mobile Systems. *Mobile*

Computing and Communications Review, 6:25-33, 2002.

- [9] See <http://www.wi-fi.org>.
- [10] See <http://www.zigbee.org>.
- [11] E. Royer and C. Toh. A Review of Current Routing Protocols for Ad-hoc Mobile Wireless Networks, 1999.
- [12] Yongqiang Huang and Hector Garcia-Molina. Publish/subscribe in a Mobile Environment. *Wirel. Netw.*, 10(6):643-652, 2004.
- [13] Gianpaolo Cugola and Gian Pietro Picco. REDS: A Reconfigurable Dispatching System. In *SEM '06: Proceedings of the 6th International Workshop on Software Engineering and Middleware*, pages 9-16, New York, USA, 2006. ACM.
- [14] See <http://zeus.elet.polimi.it/reds>.
- [15] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Design and Evaluation of a Wide-Area Event Notification Service. *ACM Trans. Comput. Syst.*, 19(3):332-383, 2001.
- [16] Elizabeth M. Royer and Charles E. Perkins. Multicast Operation of the Ad-Hoc On-Demand Distance Vector Routing Protocol. In *MobiCom '99: Proceedings of the 5th Annual ACM/IEEE International Conference on Mobile Computing and Networking*, pages 207-218, New York, USA, 1999. ACM.
- [17] Charles E. Perkins and Elizabeth M. Royer. Ad-Hoc On-Demand Distance Vector Routing. In *WMCSA '99: Proceedings of the Second IEEE Workshop on Mobile Computer Systems and Applications*, page 90, Washington, DC, USA, 1999. IEEE Computer Society.
- [18] Gian Pietro Picco, Gianpaolo Cugola, and Amy L. Murphy. Efficient Content-Based Event Dispatching in the Presence of Topological Reconfiguration. In *ICDCS '03: Proceedings of the 23rd International Conference on Distributed Computing Systems*, page 234, Washington, DC, USA, 2003. IEEE Computer Society.
- [19] Lars Aalkjær Jensen and Søren B. Madsen. Fleet Support System. Master's Thesis, Vitus Bering, Center for Videregående Uddannelse I-73, 1999.
- [20] Sørensen, C. G., & Thomsen, F. T. (2006). Synopsis regarding specific and general requirements on fleet management within arable farming. www.robocluster.dk, 1-26.
- [21] Sørensen, C.G. 2003. A Model of field machinery capability and logistics: the case of manure application. *Agricultural Engineering International: CIGR eJournal*, 5, 2003.
- [22] SAP AG 2005. Providing collaborative business solutions for all types of industries and for every major market. Waldorf, Germany
- [23] Inc. Sun Microsystems. JXTA Technology – Overview. See <http://sun.com/software/jxta/>.