Lincoln David Nery e Silva

A Scalable Middleware for Structured Data Provision and Dissemination in Distributed Mobile Systems

TESE DE DOUTORADO

DEPARTAMENTO DE INFORMÁTICA
Programa de Pós-Graduação em Informática

Rio de Janeiro
May 2014

Pontifícia Universidade Católica do Rio de Janeiro

Lincoln David Nery e Silva

A Scalable Middleware for Structured Data Provision and Dissemination in Distributed Mobile Systems

TESE DE DOUTORADO

Thesis presented to the Programa de Pós-Graduação em Informática of the Departamento de Informática, PUC-Rio as partial fulfillment of the requirements for the degree of Doutor em Informática

Advisor: Prof. Markus Endler

Rio de Janeiro
May 2014

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO

Lincoln David Nery e Silva

A Scalable Middleware for Context Information Provision and Dissemination in Distributed Mobile Systems

Thesis presented to the Programa de Pós-Graduação em Informática, of the Departamento de Informática do Centro Técnico Científico da PUC-Rio, as partial fulfillment of the requirements for the degree of Doutor.

Rio de Janeiro, May 21th, 2014

Lincoln David Nery e Silva

He received the B.S degree in Computing Science from Federal University of Paraíba (UFPB), in 2005, and M.S. degree in Informatics from Federal University of Paraíba (UFPB), in 2008.

Bibliographic data

Silva, Lincoln David Nery e

A scalable middleware for structured data provision and dissemination in distributed mobile systems / Lincoln David Nery e Silva ; Advisor: Markus Endler. – 2014.
110 f. : il. ; 30 cm

Tese (doutorado)–Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática, 2014.
Inclui bibliografia

1. Informática – Teses. 2. Computação móvel. 3. Ciência do contexto. 4. Middleware para comunicação. 5. Difusão de informação de contexto. I. Endler, Markus. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CDD: 004

to my grandfathers, Leônidas and Nicholson (*in memoriam*)

Acknowledgments

I would like to thank all my professors for all the hard work and dedication. From my neighborhood elementary school principal and philosophy teacher, Francisca Rolim, who taught me to have critical sense and always push myself to higher limits. My secondary school teacher, Marcus Varandas, who introduced me to my first programming language and made me love Informatics. Guido Lemos, for his friendship, wisdom and for give me the first opportunity to work in the academy, which is the reason I am a Ph.D today. To my advisor Markus Endler, the most patient, comprehensive, dedicate friendly and wise person I have met in the last years – really, thank you Markus, you are a saint!

To all my friends from LAC, PUC-Rio and Rio, for the friendship, support, help and availability to help me during all this years of work: thank you, guys. To my friends from UFPB, Lavid and João Pessoa that even by distance were always worried, helping and being my friends – even for asking too much how the thesis was.

I would like to explicitly mention my two dear friends, Kelly and Tati, for all the prayers and constant incentive to make me never give up. Also to all my friends who were always happy with every little new step in the conclusion of this thesis and, of course, saying their prayers in their own way and religion. Thank you guys, sure it helped! As for every thing conquered in my life, I am sure it was only possible by the constant care, strength and love I always receive from God.

To my parents, brothers, grandparents and all family, that even without an exact comprehension of the nature and the demands of my Ph.D, gave support and care. In particular, I would like to thank all the prayers of my worried mother.

And, of course, to all financial support I received from my graduation in a federal university to the scholarship provided by CNPq, essential to make the conclusion of this thesis possible.

Abstract

Silva, Lincoln David Nery e; Endler, Markus. **A Scalable Middleware for Structured Data Provision and Dissemination in Distributed Mobile Systems.** Rio de Janeiro, 2014. 109p. DSc Thesis - Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Applications such as vehicle fleet monitoring and logistic systems, emergency response coordination, environmental monitoring or mobile workforce management, employ mobile networks as means of communication, information sharing and coordination among a possibly very large set of mobile nodes interconnected by a Wide Area Network (WAN). The majority of those systems thus requires real-time tracking of the mobile nodes context information, interaction with all participant nodes, as well as means of adaptability in a very dynamic scenario, where it is not possible to predict when, where and for how long the nodes will remain connected. Despite being a subject of much research, current solutions still lack essential features required for communication with mobile nodes, such as reliable message delivery, handover support, resilience to intermittent connectivity, IP address changes and firewall transversal. This thesis proposes a data management model that enables deployment of a network of Data Provider components with reliable and on-time dissemination and transformation of information among thousands of mobile nodes interconnected through wireless internet. Performance tests indicate that our model scales to thousands of mobile nodes and supports reliable, high throughput and on-time data dissemination between several thousands of mobile Data Providers and Data Consumers.

Keywords

mobile computing, context-awareness, communication middleware, context information dissemination

Resumo

Silva, Lincoln David Nery e; Endler, Markus. **Um middleware escalável para provisão e disseminação de dados estruturados em sistemas distribuídos móveis.** Rio de Janeiro, 2014. 109p. Tese de Doutorado - Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Aplicações para o monitoramento de frotas de veículos e sistemas de logística, coordenação em situações de emergência, monitoramento ambiental ou de gestão de força de trabalho móvel podem usar redes móveis como meio de comunicação, troca de informações e de coordenação entre um número possivelmente grande de nós móveis interligados por uma rede WAN. A maioria desses sistemas requer o monitoramento em tempo real das informações de contexto dos nós móveis, interação com todos os nós participantes, bem como meios de adaptação num cenário muito dinâmico, onde não é possível prever quando, onde e por quanto tempo os nós permanecerão conectados. As soluções atuais ainda não têm recursos essenciais necessários para a comunicação com os nós móveis, tais como a entrega confiável de mensagens, suporte a *handover*, resistência a conectividade intermitente, mudanças de endereço IP e firewall transversal. Esta tese propõe um modelo de gestão de dados que permite a implantação de uma rede de componentes de provedores de dados com disseminação e transformação rápida e confiável de informações entre milhares de nós móveis interligados através de internet sem fio. Os testes de desempenho indicam que o nosso modelo consegue escalar para milhares de nós móveis e suporta disseminação confiável, rápida e com alta taxa de transferência da informação entre milhares de provedores de contexto e consumidores de contexto móveis.

Keywords

Computação Móvel; Ciência do Contexto; Middleware para Comunicação; Difusão de Informação de Contexto;

Summary

List of Figures

List of Tables

1
Introduction

Advances in mobile communication, GPS positioning and sensor technology networks are some of the driving forces that push computing to mobile-networked systems, enabling new services and applications. Many current distributed applications such as transportation and logistics, emergency response, environmental monitoring, homeland security or mobile workforce management, employ mobile networks as means of enabling communication, collaboration and coordination among the mobile nodes – which might be people, vehicles or autonomous mobile robots [8]. With the rapid increase of embedded mobile devices many of such applications are faced with the challenge to support several thousands of nodes, enabling both real-time tracking of their context/location information, and also efficient means of interaction among the mobile nodes [9], allowing the dissemination of wide interest information (e.g., emergency and traffic alerts, weather conditions, etc.) and direct communication between two nodes or a sub-set of all nodes for private communication (e.g., text messages). Moreover, in many of the systems the set of mobile nodes connected to the network can vary constantly, as nodes may join and leave the system at any time, either due to application-specific circumstances or because of intermittent wireless connectivity. Such large-scale mobile applications thus require a scalable communication infrastructure that supports reliable and on-time data dissemination among large sets of mobile nodes/devices. For example, in emergency response applications, emergency situations must be rapidly detected and members of the rescue team be quickly informed about its occurrence, as well as the actions that should be taken to lessen any possible effect. Moreover, the system must have monitoring and dynamic adaptation capabilities that enable automatic adjustment of the infrastructure services to the very dynamic data communication load caused by the mobile nodes. To ease the development of such applications, it is essential to use a middleware that provides some data management services, which will eliminate from the application developer most of the burden code related to data acquisition, transformation and dissemination.

It is important to note that despite their recent evolution, mobile devices still have stringent resource limitations when compared to stationary nodes, notably, limitations of energy supply (i.e., battery) and network reliability. So, middleware for distributed mobile applications may have additional mechanisms to handle those limitations.

Middleware to support communication in distributed mobile applications are well known and are being explored by research projects for many years [1, 2, 3, 4]. Using such systems it is possible for the participant communicating nodes to share data among them with some guaranties like fast, reliable and/or secure dissemination.

A specific type of data the nodes may share is their *context information*, which may be any information representing the actual context of the device running the application (e.g., CPU usage, free memory status, network speed, geographical location, or other sensor data), of the user that owns the device (its gender, age, profile or preferences) or the environment where the device is located (such as the environmental temperature, pressure, noise level, time, nearby nodes, etc.). The applications that share this type of information are known as *Context Aware Application* [5] and they usually use those data to adapt its behavior according to the current context of the user device at the time of its execution. For example, an application (Figure 1) could use the user food preferences, combined with the user's location, date, time of the day and weather conditions to suggest nearby restaurants for lunch. This application can also check for reviews posted by others users to make a choice, using previously shared information. There are many examples and scenarios exploring context-aware applications [6, 7].

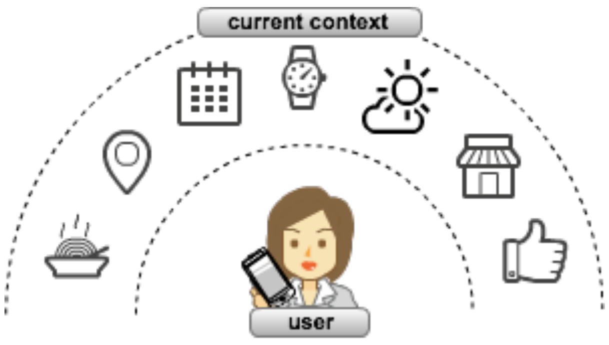


Figure 1 – Context-aware mobile application example.

1.1. Motivation and Scenario

A common characteristics of the distributed applications considered in our work is the fact that the mobile nodes periodically produce some data about their state and/or their surrounding environment (i.e., its context), as for example, their position, speed or ambient temperature and disseminate this data so that it can be processed or visualized by other nodes – both stationary or mobile ones. We also assume that each mobile node has some wireless network interface that is capable of running the IP protocol, which in fact, is the case for most current wireless and mobile networks. Examples of such applications include on-line driver assistance, fleet monitoring and management, remote car monitoring services, or police task force command and control systems. In several of such applications the mobile nodes may also receive commands/instructions from remote nodes (e.g., a Control Center or even other mobile nodes) that influence its operations state or its future mobility pattern. For all these applications, the main requirement is that, as long as the mobile node has some connectivity, data and messages produced by it must be reliably transferred and disseminated *on-time* to all interested nodes, i.e., with minimum possible delay[1].

In our work, most mobile nodes we are considering are connected vehicles. However, note that connected vehicles are a vast field of research, with a number of technologies associated with it [10, 11, 12, 13]. Nevertheless, in our work we assume a connected vehicle is any vehicle that carries a connected mobile device, which may be a smartphone, tablet or any embedded device. Many are the benefits a vehicle can leverage while connected. For example, if vehicles are periodically sharing their location, they can infer traffic conditions and choose better routes to avoid traffic.

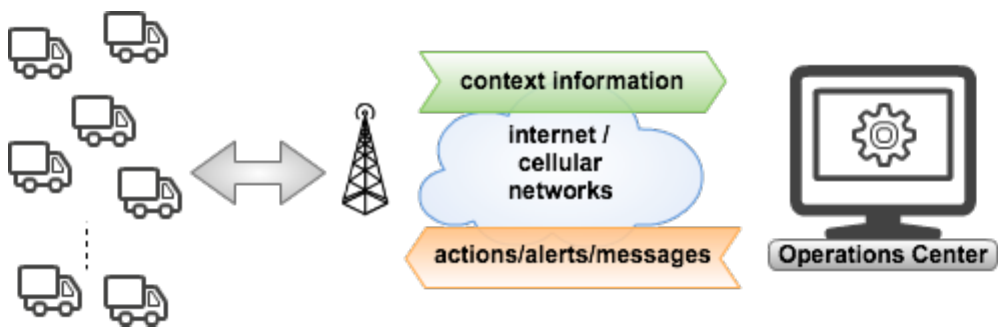


Figure 2 – Scenario illustration picture.

1 This minimum delay, in fact, is largely determined by the latency of the wireless connection.

Our scenario is defined by a concrete Mobile Vehicle Tracking and Management system to be used by a major Brazilian gas distribution company, which operates throughout the entire country. Through this system (please see Figure 2), the company's Operations Center application should be able to track the trajectories of its trucks in real-time, in order to optimize the trucks' itineraries, to detect and notify drivers about obstructions or traffic jams on roads, to predict and minimize delays, detect and notify truck drivers about bad weather conditions, and to monitor individual driving behavior (e.g., elapsed time on both planned and involuntary stops, detours, high-speed driving, frequent acceleration and de-acceleration, and others). In order to monitor a single driver’s driving behavior the application just needs that single truck’s context information data. However, to detect traffic jams or weather-related driving conditions (e.g., slippery roads, fog, etc.), the context information data from various trucks (and even from roadside sensors) must be combined and analyzed to infer such higher-level information – e.g., the location variation in time of trucks on a road could be processed to calculate its speed, which can indicate a traffic jam if that velocity is below a defined threshold. Furthermore, all the data must be quickly processed and combined with other information data corresponding to the same period of time to avoid inference of wrong information (e.g., using old speed information detecting an inexistent traffic jam). Moreover, the application must also support communication with drivers – to send them instructions or alerts – both individually and to subgroups of drivers. These groups of mobile nodes (i.e., the trucks) are typically determined by the geographic region in which the nodes are currently located, or by any other context-based criteria. For communication with the drivers/vehicles, the company will use several cellular operators, since in each region of the country there are significant differences of connectivity quality and coverage among the operators, especially in remote areas. Thus, it is expected that the mobile nodes may obtain several IP addresses along their journeys, may experience temporary data link disconnections, and that the links have to cross the cellular operator-specific firewalls.

The company’s current fleet has almost 10,000 trucks and this number grows every year. Additionally, there are other third-party transport companies that serve the gas company, and whose trucks it must monitor and manage as well, which increases the total number of trucks to be managed by the application by almost 90,000 trucks. In spite of such large numbers, the system must enable real-time monitoring (i.e., on-time dissemination of context information data), fast and reliable detection of abnormal situations, as well as scalable communication services. This implies that no context information or application message should be missed, i.e., communication should also be reliable.

So, the described scenario presents the problem of how to perform a reliable and scalable on-time dissemination of data among thousands of producers and/or consumers mobile nodes, distributed across several networks (including wireless networks), with no guarantee of availability of their connectivity.

1.2. Contributions

Much research has been done in communication middleware for large-scale applications, but only few support large-scale mobile networks with QoS guarantees for the mobile communications and data dissemination, in particular reliable and low latency message delivery over the wireless connections.

In this thesis we will present some existing communication middleware and models for large-scale applications supporting mobile nodes. We will then propose a new model capable of reliable and on-time dissemination data for mobile applications, even in environments with limited wireless network capabilities – this is our hypothesis. Our data management model will be implemented as a middleware and be evaluated with respect to its scalability and communication reliability features. Then, the main contributions of this thesis are the following:

1. Describe the features and implementation of a Mobile Reliable UDP protocol for mobile nodes which transparently handles short-lived temporary mobile node disconnections and ensures reliable packet delivery across these intermittent disconnections;
2. We present a communication middleware model and implementation, give evidence of its scalability, and show how it supports efficient and reliable unicast, *groupcast* and broadcast message delivery to mobile nodes in spite of IP address changes, temporary disconnections, and Firewall/NAT traversal by using our MR-UDP protocol.

The thesis is organized as follows: after the introduction in this chapter, in Chapter 2 we will present state-of-art systems for large-scale mobile communication. In Chapter 3 we present and discuss related work and their applicability for our proposed model. Our proposed model is specified in Chapter 4. In Chapter 5 we describe the implementation of our model in details and present proof-of-concept application and several performance tests and evaluation in Chapter 6. Finally, in Chapter 7 we present our conclusion remarks and points to future works.

2 State of Art on Large-Scale Mobile Communication

A lot of research already has been done for large-scale communication on distributed mobile applications. As pointed in [14] the Internet has considerably changed the scale of distributed systems, which now may involve thousands of entities widely distributed all over the world. Such distributed systems are intrinsically heterogeneous in many aspects. Usually, nodes are running on different devices, running different operating systems, different software versions, connected to different networks, static or mobile, and must seamlessly be integrated in the communication network despite any specific characteristic or limitation. To cope with such large-scale communication in a heterogeneous environment we need systems that provide mechanisms well adapted to this situation.

2.1. Publish-Subscribe Systems

It is widely agreed that Publish-Subscribe systems are well suited to be used in systems that deal with the sharing of data among mobile nodes. Due to its loosely coupling, it is well tailored to communication in large-scale distributed applications [14].

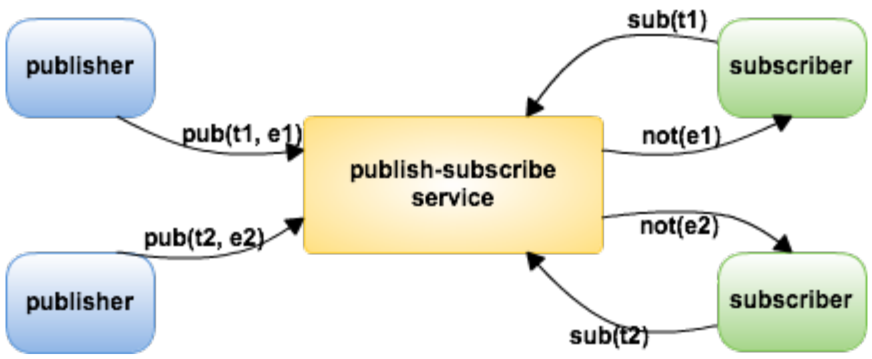


Figure 3 – Publish-Subscribe communication paradigm.

A publish-subscribe system is characterized by the asynchronous exchange of messages (*events*) between distributed nodes – which may be mobile nodes. The main mechanics of a Publish-Subscribe system is depicted in

Figure 3. With the help of a middleware service, consumer nodes (commonly called *subscribers*) indicate their interest in some kind of information type through a *subscription*, where usually some filtering expression – or criteria – is passed as an argument of the subscription operation. A subscription may be undone by an *unsubscription*. The nodes that produce data items (*events*) are called *publishers* and the middleware service is responsible to deliver (*notify*) *new* events to all the subscribers whose filtering expression (subscription criteria) matches the attributes of the published data items.

This communication paradigm promotes the decoupling between publishers and subscribers by the middleware service, which can be decomposed in three main dimensions:

- *Space decoupling*: The publishers and subscribers do not need to know and to reach each other. It is the middleware's task to guarantee that publications matching subscriptions reach the interested subscribers. So, publishers do not need to know which subscribers are receiving its events, and vice-versa. This means that no node needs to hold any reference or direct connection to any other node, which is a good feature when dealing with a large and dynamic set of nodes.
- *Time decoupling*: The parts involved in the communication do not need to be active at the same time to communicate. The subscribers may be temporally disconnected and after reconnection should receive all pending notifications. At the same time, when a notification reaches its destinations, the actual publisher of the events may be disconnected itself. Thus, time decoupling supports asynchronous communication, which is very suitable in mobile environments, where the communicating nodes may join and leave the network at unpredictable moments.
- *Synchronization decoupling*: The interaction between parts is asynchronous and does not need to be instantaneous. After a publication, the publisher continues its execution and the publish-subscribe system will notify all the corresponding subscribers when possible. Similarly, the consumption of the data item by the subscriber may occur asynchronously, by receiving a notification while executing any parallel activity. This is interesting in a large-scale communication because it separates the communication process from the normal application execution itself, so the

communication process does not interfere on the actual application workflow.

This decoupling between the production and consumption of the information also helps to increase the scalability, since the parts are only responsible for the publications and subscriptions, leaving all the hard work of matching the published data items with the subscription criteria (filtering expressions) to the middleware.

A Publish-Subscribe system can be classified in three categories. Each one presents some differences in regard to the supported subscription (and notifications), but all follow the general idea of decoupled communication explained so far [14].

In a ***Topic-Based*** publish-subscribe the subscriptions are made by designating a topic which the subscriber is interested in receive notifications. Usually the topic is represented by a *keyword*. When a publication is made to the same topic, all subscribers are notified. For example, a subscriber indicates its interest in the topic *"photos"* and once a new photo is published on this topic, the middleware will deliver it to all subscribes. In practical terms, a *topic-based* boils down to creating a specific tagged communication channel for every keyword (i.e., topic).

Content-Based Publish/Subscribe systems do not matches its publishers and subscribers by a tag on the data, but by some properties of that data. For example, a subscriber may only be interested in photos of wild animals. The use of a language for expressing the subscription (or filtering) criteria is essential in this kind of communication, so that at subscription one can express what information is needed. The matching is not as trivial as in the topic-based variant, and the more powerful the language is, more computational expensive will it be to check for the matching subscriptions at every publication. While in the topic-based variant all publications in the same topic cause notifications to be delivered, in the content-based case, every publication must be checked against every subscription expression.

Yet another variant is the ***Type-Based*** Publish/Subscribe, which extends the topic-based scheme by defining the type of publication that is desired. For example, the photos topics may involve many picture formats (e.g., jpg, png and others), but some subscriber may have only the rendering capability for just the jpg type, and so this type will be informed on the subscription.

Because of it's decoupled nature, the publish-subscribe communication is well suited for mobile communication. Using mobile nodes as publishers and subscribers, all the computational expensive and resources consuming communication and subscription matching process will be done by the publish-subscribe middleware's infrastructure, which typically runs on servers in a cloud, a cluster or a private backbone network. The mobile nodes will only use their resources when there is need to publish a data item or when a notification is received. As already mentioned, they do not even need to be always active (or connected) to have its publications disseminated or notifications received, since the middleware will intermediate all communication asynchronously.

Usually, the publishers produce new events with some interval between publications, i.e., their basic behavior (and most middleware implementations) does not focus a constant flow of events, which is the case of the dissemination of context information for our proposed model in the previous chapter. However, some research already identified this need and proposed some solutions [14, 15, 16].

However, despite the huge number of research and available implementations, not all publish-subscribe systems are specifically tailored to mobile communications and mobile networks. Some systems may offer some Quality of Service (QoS) guarantees, but, in most cases, these guarantees are given only for the interaction among stationary nodes interconnected through a reliable and high-performance wired network infrastructure [17, 18, 19].

2.2. DDS Systems

Much research has been done in publish-subscribe, but only few support large-scale mobile networks and at the same time offer some QoS guarantees for the mobile communications, specially reliability and low latency message delivery. On the other hand, OMG's Data Distribution Service for Real-time Systems (DDS) standard [20] offers high performance communication capabilities, and is currently used for several real-world distributed mission-critical applications.

DDS specifies a fully decentralized (peer-to-peer), scalable middleware architecture for asynchronous, publish-subscribe-like data dissemination, supporting several Quality of Service (QoS) policies, like best effort or reliable communication, support for *late joiners*, data flow priorization, and several other message delivery optimizations, etc. [21]). Unlike other publish-subscribe systems, DDS provides explicit control over the communication properties and most efficient use of the available network resources, through its QoS policies and fine-tuning of its underlying routing services (e.g., its QoS policies deadline, latency budget or transport priority, etc.), that are critical for high performance and almost real-time communications. Hence, since it combines the asynchronous communication paradigm with efficient network usage and guaranteed message delivery policies, DDS should be useful also for large-scale mobile applications.

DDS specifies the Data-Centric Publish-Subscribe (DCPS) model, which defines standard interfaces that enable applications running of heterogeneous platforms to read/write data from/to a shared global data space. The DCPS model is the core of a DDS system. Applications willing to publish some data should declare its intent in that global data space, specifying the topics of interest that are related to the data to be produced. Similarly, applications that want to subscribe to some data, uses the same data space to declare their intent to receive notification on specifics topics. The underlying DCPS middleware propagates data samples written by publishing applications into the global data space, where they are disseminated to subscribing applications [20]. The DCPS model decouples the declaration of information access intent from the information access itself [23], thereby enabling the DDS middleware to support and optimize QoS-enabled communication.

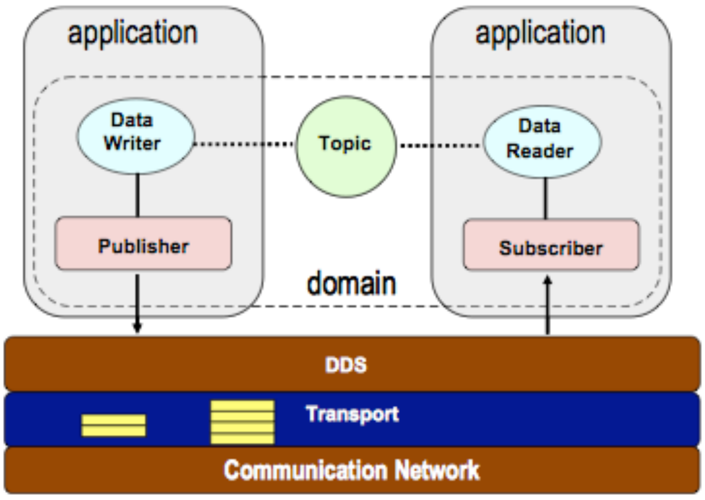


Figure 4 – Architecture of DDS. Source: [22]

As shown in Figure 4, a typical DCPS model is comprised of the following entities that provide functionalities for a DDS application to publish/subscribe data samples of a certain topic of interest [22].

- **Domain**. A domain is a virtual data (and name) space that connects publishing and subscribing nodes of applications. Only applications within the same DDS domain can communicate. This helps to isolate and optimize communication within a community that shares common interests.
- **Publisher/Subscriber and DataWriter/DataReader**. A Publisher is a factory that creates and manages a group of DataWriter entities with similar QoS policies. A Subscriber is a factory that creates and manages DataReader entities. DataWriter/DataReader entities are the actual data objects required for application to publish/receive data samples.
- **Topic**. A topic connects a DataWriter with a DataReader. Data flow can happen only when the topic published by a DataWriter matches the topic subscribed to by a DataReader. Communication via topics is anonymous and transparent, i.e., publishers and subscribers need not be concerned with how topics are created nor who is writing/reading them since the DCPS middleware manages these issues.

However, in spite of its advantages, DDS cannot be efficiently deployed neither directly on mobile nodes nor in wide-scale wireless networks or WAN. The main reasons are its extensive use of IP multicast in DDS domains, the lack of proper mechanisms to handle intermittent connectivity and IP address variability, and the problem that resource-limited (mobile) devices cannot function well as DDS peers, since they must cache and route data for other peers.

During the development of our work, we have made some tests and concluded that the main DDS implementations available (we mostly used CoreDX[2] and OpenSplice[3]) show very good high-performance topic-based communication in a private and well-controlled and configured network [24]. The support for communication between two LAN is precarious and promotes the lost of some QoS guarantees, as pointed in [25]. CoreDX has one of the best mobile implementation, however, the mobile node must be in the same LAN of the other nodes. Consequently, DDS cannot be used in a large-scale mobile communication scenario distributed over many mobile networks.

2 http://www.twinoakscomputing.com/coredx

3 http://www.prismtech.com/opensplice

For our scenario, we envision a communication middleware that could provide the best of publish-subscribe and DDS systems for the communication of mobile nodes. We would like the decoupling nature of publish-subscribe with the QoS guarantees of the DDS to work with thousands of mobile nodes distributed over many mobile networks presenting intermittent connectivity.

2.3. Context Management

Context is "any information that can be used to characterize the situation of an entity (person, place, physical or computational object) that is considered relevant to the interaction between the entity and application." [1] The context information is primarily used to adapt a context-aware application’s behavior according to the actual situation at the moment of its execution. For example, a personal meal assistance application could list food options from restaurants nearby the device running the application, taking into account the time of the day (i.e., food appropriate for breakfast, snacks, lunch or dinner), the user preferences (which can be configured by the user or learned by the choices that were made from previous suggestions) and quality ratings made by others users of the application for each meal option. However, context-awareness is not used only for adaptation. Sports tracking [26] and healthcare monitors are example of applications that monitor the user’s context (e.g., his/her state of activity and well being) but may not have adaptable behavior.

The scenario presented in Chapter 1 has shown that the trucks and the Control Center might exchange any type of structured data, including application-specific and as well as context information (e.g., trucks’ location, speed, fuel level, etc). Since in mobile systems the communication (and processing) of structured data carrying context information has specific performance requirements on the dissemination and processing infrastructure, it is important do look at the required features of such systems and propose an embracing solution.

In order to implement a context-aware application it is the programmer’s responsibility to integrate the application’s main logic with the code to probe the local sensor and read raw data, functions to interpret the this raw data to extract useful information (e.g., detect some pattern in the sequence of probes), perhaps combine the data from multiple sensors to infer higher-level information and implement the application behavior switch according to any expected context the application may meet while executing. Moreover, if the sensor is available in another connected remote device, then network communication code should also be coded. In many context-aware applications all this code is still interweaved with the business logic of the application itself, which is not desirable in terms of Software Engineering practice that recommends modular programming and separation of concerns. The increased availability of sensor-rich mobile devices (e.g., smartphones with embedded sensors, and many sorts of networked sensors deployed in the environment, etc.) has as consequence a high heterogeneity of sensor types and devices, which in turn can make the implementation of context-aware applications an even more challenging task. For example, two different smartphones models (even of the same make) may have different GPS sensors and different APIs to probe the same context information, depending on which version of the operating system is installed.

Recurrent requirements for context-aware application are identified in [27]:

- Handle heterogeneity by dealing with different sources of context, e.g., sensors, Web Services, nearby devices, user, etc.;
- Specify the relation and dependencies between different types of context information and how they depend on each other;
- Handle the sensor data imprecisions and model the confidence level of context sources;
- Support some reasoning on context information or events, for detecting specific situations or to infer higher-level context information;
- A formal notation to describe all relevant context information, so as to facilitate the integration between applications and sensors;
- Provide efficient context information provisioning, making the information easily available to the applications.

Therefore, to implement a large-scale context aware system that supports the dissemination of context information from/to thousands of mobiles nodes is a demanding challenge (see next section). Knappmeyer et al. [28] summarized the above requirements for such systems in two essential parts:

- The **Context Management** subsystem, responsible for the context acquisition and dissemination; and
- Context Modeling, concerned with manipulation, representation, recognizing and reasoning about context and situations.

Thus, the use of a middleware that provides context management services makes the implementation of context-aware application much simpler by letting the programmer focus only on the application-specific code [29]. Such middleware usually provides abstractions and mechanisms for the acquisition of context from sensors, transformation of context data and the dissemination of produced context to all interested nodes, as we will show in the next chapter.

2.3.1 Context Data Dissemination

As defined in [30], context data dissemination (or distribution) is the task of delivering any relevant context data about the environment of one mobile node to all interested entities of a distributed context-aware application. Context data distribution is a critical function of most context aware systems. On the one hand, context data has to be delivered timely to let systems promptly adapt to the current context. On the other hand, the middleware should transparently manage and route large amounts of context data sensed by the mobile nodes through the system, while ensuring on-time delivery of these data to the interested parties. And this has to happen in spite of intermittent connectivity and non-negligible communication latency of wireless connections, thus hindering both system scalability and reliability. The importance of context data dissemination has been shown by several research projects and emphasized by several surveys about middleware for context-aware systems [31, 32, 33, 34, 35].

3
Related Work

In this chapter we discuss related works that address the problem of large-scale dissemination of data/context information for mobile nodes. We first define the main features required for achieving data dissemination in large-scale mobile systems, such as in the terms of the presented scenario. After the individual description and evaluation of the works we then present a comparison table based on the specified features. We focused on related work presenting solution for context information dissemination because most of the data that is shared in our scenario are context information. Moreover, those systems is more likely to present a more embracing approach and concerns with mobile communication, as will be detailed in the following sections.

3.1.
Desired Features

The scenario presented in the first chapter brings some features/requirements that should be addressed in the models and middleware systems intended for large-scale data context dissemination to/from mobile nodes. We will separate these features in network-related, on one hand, and in mobile device- and connection-related, on the other hand.

The network infrastructure that enables communication among mobile nodes may be constituted either by fixed infrastructure-based service providers (mobile operators or private networks) with a set of dedicated servers in a backbone *core network*, or, else, by direct peer-to-peer links among the mobile nodes and an *ad-hoc* routing algorithm. Due to the fast growth of the Mobile Internet, in most cases a fixed infrastructure is used. It seems that the lack of guaranties of mobile communications (e.g., unreliable and intermittent connection) can only be well addressed by the use of fixed infrastructure services. Moreover, the management of a peer-to-peer network of thousands of mobile nodes can be very cumbersome or even impossible. In fact, most large-scale mobile applications require a middleware that provides network communication services with the following features:

- **Reliable communication**, meaning that all the data generated should reach all interested active (i.e., currently connected) nodes. However, data generated while a node was not active can be missed, since some information lapses, as will be detailed in the following;
- **On-time dissemination of context information**. For much context-aware applications it is required that the information received is accurate, meaning that adaptation should be done on the basis of the freshest context information, as in many cases, information representing a outdated context are useless [30]. Therefore, all publications of context information should reach interested nodes with the minimum possible delay, i.e., a time close to the common delay noticed when using the actual network technologic being used; and
- **Scalability** (in terms of the number of mobile nodes): the overall network performance should not be significantly affected when the number of nodes increases – at least the performance should not be affected in the same proportion. Additionally, the increase in the number of nodes should be compensated by the provisioning of additional resources on the network infrastructure, which should maintain the overall performance similar;

To effectively provide the above features to mobile nodes, the intrinsically characteristics of such devices and its connectivity should be directly addressed. Despite their recent evolution, mobile devices still have stringent resource limitations when compared to fixed nodes, notably, limitations of energy supply (battery) and network reliability. So, it is expected that models and middleware for distributed mobile applications may have additional features to compensate those limitations. We list some of them below:

- **Support for dynamic connectivity configuration**, meaning that the system should be prepared for unpredictable connectivity status of the mobile nodes, that could join and/or leave (for good) the network at any time and with no warnings;
- **Support for intermittent connectivity**, which means that temporary (short-term) disconnections should be transparent to the applications. It is desirable that the system is capable of hiding such short-lived disconnections from the applications that should be notified only when the disconnection exceeds a certain time

threshold. Moreover, any message sending that was dispatched during the disconnected period should be properly delivered after a new reconnection;

- **IP address change identification**, especially when using mobile networks, since every time the data link connection is broken and reestablished the cellular provider assigns a new IP address. When the node reconnects to the network, this should be detected and the connection should be re-established, with any on-going communication transparently resumed;
- **Support for Firewall traversal** – means that mobile nodes behind firewalls, which is the case in most mobile networks, should be reached for direct data communication at any time, even if they do not have public IP addresses. It is a important requirement, since in most mobile networks it is given private IP addresses to the mobile devices.

3.2. Discussion on Related Work

Several well-known research works have proposed models for context management in mobile networks; some of them also provide middleware implementations. In this thesis we focus on works that provide at least some of the features described in the previous section and discuss their main strengths and drawbacks [28, 36, 43, 44, 50].

Aiming in providing a global-scale infrastructure for context-awareness to be shared simultaneously by many applications, Nexus [38] proposes the integration of millions of context producing nodes (e.g., sensors) that produce an arbitrary number of different kinds of context information and subscriber nodes. The supported nodes are assumed to be heterogeneous, such as networked sensors, mobile applications or static nodes. A federated network of servers that are specialized to manage one type of context information each provides means of *scaling the system,* in contrast to centralizes architectures. In Nexus, each server is specially developed to handle the characteristics of this type of information, such as location, temperature, map, etc. This specificity of the servers makes the addition of new types of information not trivial, since it requires a new server to be designed, implemented, deployed and its APIs distributed to the application developers, before the context information can be used by the applications. Moreover, context information is accessed through queries submitted to server handling the desired information, which makes the server behave much like a database of context information. This characteristic puts into question Nexus' *scalability*, since a type of context information that entails a high volume of produced data items and which may be consumed by many nodes (e.g., geographic coordinates of thousands of nodes in a tracking application) could make that server a centralized bottleneck and thus compromise the system's scalability. Also, this *synchronous,* query-reply-based communication mode does not support any decoupling between the servers and the consumers of context information. Asynchronous notifications about new bits of context information were still "being designed".

The flow of transformations starting from raw sensor data until it becomes useful context information and reaches the application can be specified in XML, which is the approach adopted by Solar [39, 40]. The middleware enables the construction of what it calls a *Context Fusion Network* (CFN), a *planetary* network composed of so-called *planets*, where each one is responsible to execute a single transformation step over the data/information flow (i.e., filter, aggregation, etc.), which is defined in XML. By this, Solar supports *expansibility* of context processing and supposedly *heterogeneity*, since any platform can potentially handle XML. After visiting several planets, context information becomes accessible to consuming nodes through *asynchronous* communication. The communications between planets are handled by Pastry [41], a peer-to-peer *scalable* communication substrate based on Distributed Hash Table. The Solar middleware also supports *dynamic adaptation* of the planetary network, which can rearrange and deploy new planets to help maintain a well-balanced distribution of processing load among planets and to reuse already deployed planets for multiple transformation graphs. Mobile devices are handled by *proxy-planets* in the Pastry network. They cache data sent to/from the mobile devices when the device presents *temporary disconnections* or during a *handover* to another *proxy-planet*, and deliver any non-acknowledged communication after connectivity is re-established.

Mobile devices are also the main subjects of the work in [46, 47], which presents a completely *decentralized architecture* composed of federated brokers that communicate in peer-to-peer, *asynchronous* mode. This decentralized and decoupled approach promotes *scalability*. A publish-subscribe API is provided to the applications. The routing mechanism is quite complex, and requires routing tables to be exchanged between and managed by the brokers. Such routing table includes every subscription, including filters, which are evaluated once every publication is generated by both the producer and the brokers, *reducing* performance and *scalability*. Mobility is handled by providing a mobile broker deployed at the mobile device, which has essentially the same functions as the brokers deployed in the fixed network, but are used exclusively by the mobile applications. *Intermittent connectivity* is *not* directly handled by the system and an error message is returned to the client application when a context producer tries to publish information while the device is temporary disconnected, so it needs to explicitly try again later. Furthermore, running a broker o a mobile device with all its tasks, including subscription-filtering tests, should reduce the overall performance and overload the mobile device's resources.

LoCCAM [48] is a middleware for Android[4] (hence, *no platform heterogeneity*) that presents an architecture that implements a decoupling between the component layers responsible to produce and consume context information and the network communication. The middleware relies on already established middleware solutions to build a concise framework: OSGi[5] is used to implement the *ContextAquisitonComponents*, which makes *expansibility* possible since it allows the deployment of new components at runtime, and Linda [49] is the network middleware, used for communication in both tuple-spaces and/or publish/subscribe (*decoupled*) paradigms. The implementation *does not use* much of the *device resources*. Despite targeting mobile devices, *intermittent connectivity is not addressed*. *Scalability* is *not* mentioned, and seems to leave this responsibility to Linda.

Apparently, so far there is only few research and development on DDS-based middleware systems for mobile distributed applications in arbitrary wireless networks. Most of DDS studies present comparisons between and benchmarks of different DDS vendors' implementations, such as [51, 52, 53], but none of them mentions wireless networks or mobile DDS deployments. Among the few works that focus on mobile devices, we found the DDS-based middleware proposed in [54], named DDSS. It includes a specific architectural element that supports mobile nodes and ensures *reliable data delivery* even for mobile subscribers that switch their wireless access point during system operation (i.e., *handover support*). In the proposed architecture all mobile devices execute a lightweight version of DDS, the *Mobile DDS Client*, whereas stationary nodes on the fixed communication network run full-fledged DDS nodes and are responsible for the routing and dissemination of data to all nodes. Due to DDS' connectivity and

4 http://www.android.com

Firewall/NAT traversal restrictions (unless a VPN is created), all these Mobile DDS Clients must run in single network domain and rely on stable wireless connectivity. Moreover, the authors present no data about the communication performance over wireless networks.

REVENGE [25] is a DDS-compliant infrastructure for news dispatching among mobile nodes and which is capable of transparently and autonomously *balancing the data distribution* load in the DDS network. It implements a P2P routing substrate - deployed on a LAN - that is *fault tolerant* and *self-organizing*. More specifically, it is able to detect crashed nodes, and to re-organize the routing paths from any source node to any mobile sink nodes. Multiple copies of the deployed network infrastructure are available to increase *reliability* and *availability*. For this, the sources of news (the information publishers) must publish at each copy of the network, which therefore will use *more mobile devices' resources*. A relay-based protocol is presented to interconnect different DDS domains in the Internet or Wide-area networks (WAN), which improves *scalability*. Since all nodes run DDS (mobile nodes have the DDS minimum profile), it has full support of DDS QoS policies. REVENGE has been tested in a wireless network on an University Campus-wide wireless LAN, but the authors have *not shown* performance data in situations where the mobile nodes had *intermittent wireless connections* and *suffered IP address changes*. Concerning *asynchronous communication* capabilities at the mobile nodes, this system provides full DDS-based Pub/Sub support.

3.3. Comparison of Related Work

As shown by all the research mentioned on the previous section, there is a large amount of work focused on large-scale context/data dissemination supporting mobile devices. However, only a few of them directly deal with the limitations found in the mobile devices (specially the intermittent connectivity), or offer means for on-time delivery of notifications. In almost all the proposed middleware, the applications would have to explicitly handle such limitations; however, we think that such features should be automatized by the system. Scalability and reliability also does not seem to be a concern for many of the works, especially the ones based on simpler publish-subscribe mechanisms. The current research work that aim at extending (or using) DDS for mobile communication and context awareness is promising, but DDS has a number of

5 http://www.osgi.org

limitations already pointed [55], particularly when dealing with mobile nodes and multiple domain networks in a WAN. During our research we also encountered several of these limitations of DDS [24]. Another feature we need for our scenario is support for on-time dissemination of context information; nevertheless, none of the works specifically mention such feature.

Moreover, despite the common objective to design and implement an infrastructure for large-scale data dissemination among mobile nodes, the specific requirements and model/system characteristics of the works described in this chapter can be very different from each another. Most of them chose features focusing at very specific scenarios, like [50] with a solution for peer-to-peer/vehicle-to-vehicle communication. This makes it a bit difficult to compare the works, including the fact that most of them do not present any graphs or numbers about performance results. Thus, we chose to compare the works only according to the features discussed in Section 3.1, which we think are the most important ones for our scenario, as indicated in Table 1.

Features / Related Work	NEXUS	Solar	[46, 47]	LoCCAM	DDSS	REVENGE
Reliable Communication	Yes	-	No	-	Yes	Yes
On-time Data Dissemination	No	-	No	-	-	Yes
Scalable	Yes	Yes	Yes	-	Yes	Yes
Supports Dynamic Connectivity	-	Yes	No	-	-	-
Handle Intermittent Connectivity	No	Yes	No	-	-	-
Identifies IP Changes of Nodes	No	-	No	-	No	-
Firewall Traversal	No	No	No	-	No	No

Table 1 – Features comparison of related works.

The lack of a generic solution that addresses all – or at least most – of the features indicates that there is still much space for research in this field, that many important issues are not addressed by current works, and that yet we don't have a universal, optimal solution for scalable mobile context management and on-time dissemination. In the next chapter we present our model and argue about its applicability for the scenario described in the first chapter.

4
Proposed Model

Aiming the provision of scalable large-scale reliable and timely data dissemination for applications composed of thousands of mobile nodes that are continuously producing, sharing and processing structured data, we propose a conceptual model called *Data Sharing Network*. The model tries to overcome the limitations presented in current works regarding applicability to the scenario presented in Chapter 1.

4.1.
Model Concepts and Overview

A fundamental element in our model is the **mobile node** (MN), which can be any mobile device connected to a wireless network, such as a cellular network or a WiFi Access Point, and capable of sending and receiving data. Typically, a mobile node has some stringent limitations, like restricted battery life, inferior processing power, limited memory resources and untrustworthy intermittent network connectivity. In our model, we are considering that mobile personal devices (i.e., smartphones, tablets) will be the most typical examples of mobile nodes. However, a mobile node can also be an embedded device or a notebook. When a mobile node is a machine without significant resource limitations (e.g., a notebook), we may call it just as **node** for differentiation purposes, but from the point of view of the model, and for the sake of generality, all elements are considered **mobile nodes**.

While the mobile nodes are connected to a network, they may share any kind of data among them. However, the majority of them will be **structured data** representing some context information data, which is any information defining the current state or the device, user or its environment [1]. When context information is directly collected from a sensor, it is commonly referred to as *primitive context*. But context information, may also be produced by the combination of multiple primitive information, e.g. when processing the relative distance between a mobile node and other nodes using the geographic location of these nodes. In this example, the context information is referred as *composed*, *derived* or *aggregated*. In our model, any mobile node can be the producer or the consumer of any information, either primitive or derived. The information can be achieved by probing data from a local embedded sensor or by acquiring external information from others nodes, processing and combining them to derive higher-level information like in the above-mentioned example.

Nowadays, there exists a wide spectrum of affordable and tiny sensors that can be used to produce context information for context aware applications. Most mobile devices already have embedded positioning sensors (using GPS, network-based, or Wi-Fi *fingerprinting* technology), thermometer sensor, light sensor, accelerometer, gyroscope, noise sensor, compass and others. The process of probing and interpreting the raw data from sensors and produce useful information is specific for each one of them. For this reason, we define a specialized element called **Data Provider** (DtP), which is a software component that is deployed on a mobile node and is responsible for polling raw data from a sensor, process this data and generate the specific kind of useful information for the applications. Usually there will be one Data Provider for each type of sensor deployed in the system, and other Data Providers that convey more abstract information which is derived from the more basic raw sensor data, e.g., the devices' geographic position (abstract information) may be obtained either from a Data Provider that interacts with the GPS sensor, or by a Data Provider which uses network-based position inference.

The Data Provider's act of producing a new piece of information and preparing this information for dissemination over the network is defined as **Data Update** (DtU). Every Data Update generates an **information object**, which carries the information and is disseminated to all the nodes that are interested in that type of information, which may include the originating node itself.

When an application requires some type of structured data, it must express this interest through a **Data Consumer** (DtC), which is responsible to receive and process the Data Updates from the corresponding Data Providers. The Data Provider may not be locally available on the device running the application, for example, some application may depend on structured sensor data that Is only available from a remote node (e.g. a sensor mote, a Electroencephalogram (ECC) sensor device, etc.). Thus there must be a process of **discovery of Data Providers** so that the applications can remotely invoke and deploy their needed Data Providers on other mobile nodes. Once deployed, the Data Provider will start producing Data Updates, which will be disseminated on the network and reach the application that started this process.

Sometimes an application needs information that cannot be produced by other mobile node on the network except itself. For example, the device’s fine geographic location can only be obtained by the device’s location sensor – like its internal GPS. Still, the mobile node may not have a copy of the Data Provider that probes the embedded sensor. In fact, due to memory size limitations, it is not recommended to have a copy of all Data Providers deployed in every mobile node. Therefore, there must be a process of **deployment of Data Providers**, which stands for the sharing of the Data Provider’s software components themselves between mobile nodes, i.e., the transfer of a copy of a DtP from a node to another.

In Figure 5 we show an example of DtP deployments according to our model. The mobile nodes are represented by yellow rectangles and each one has several DtPs and/or DtCs locally deployed. DtPs for primitive context information do not consume DtU (incoming black arrows), which is not the case for derived context information DtP, which may consume DtU from more than one DtP.

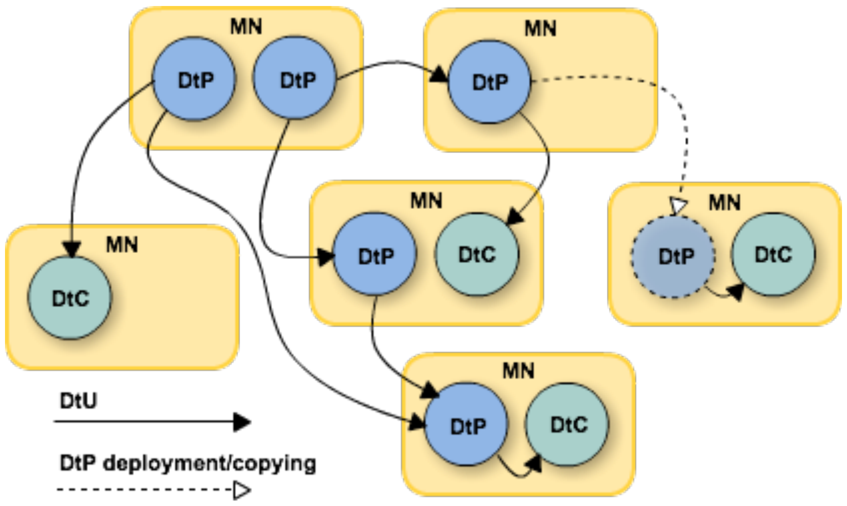


Figure 5 – Data Sharing Network model overview.

4.2. Underlying Architectural Model and Features

Complementing the model overview of the previous section, in this section we explain and discuss some underlying characteristics and features of the proposed model. Independently of implementation decisions, we argue that the characteristics and features explained in the following sections are essential to provide scalable and reliable on-time data dissemination among mobile nodes.

4.2.1. Intermediation Through a Fixed Network

Since in real world mobile applications it is impossible to predict when and for how long a mobile node will be connected, our model does not make any assumption about the mobile node’s connectivity, i.e. its long-term availability for communication. Instead, we take it for granted that any mobile node can loose its wireless connection anytime and for an arbitrary period of time, for any reason. Thus, every communication between any two mobile nodes must count on the support of a fixed network infrastructure (a “core network”), a characteristic that is also present in most related work discussed in Chapter 3. Relying only on direct and *adhoc* mobile-to-mobile connectivity can make it very difficult, or even impossible, to implement reliable and on-time message delivery in any situation.

This fixed core network will manage, route and buffer any messages to/from every mobile node until they are effectively delivered. Thus, one needs the presence of *brokers* in the fixed network to act as connection points and intermediaries for the mobile nodes. Some characteristics of the model in respect to the brokers are:

- It must be possible to deploy as many brokers as necessary to handle the number of currently connected mobile devices, since one broker may get overloaded if thousands of mobile nodes are connected to it;
- Each broker shall serve as the communication hub for a set of mobile nodes, being and be responsible for intercepting and forwarding any messages addressed to the mobile nodes connected to them;
- The mobile nodes must maintain the connection with their broker open all the time, so that it is possible to reach the mobile nodes as soon as new messages are sent to them;
- The broker must intercept and translate the messages from the protocol used for the mobile connections to the protocol used in the fixed network, and vice-versa.

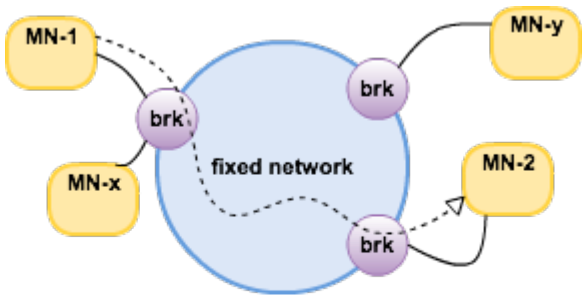


Figure 6 – Message exchange between mobile nodes through brokers on a fixed network.

Figure 6 illustrates mobile nodes connected with their brokers (brk). In the figure the mobile node *MN-1* sends a message to mobile node *MN-2*, which is forwarded by MN-1's broker – through the fixed network – to *MN-2's* broker, which then forwards the message to *MN-2*.

4.2.2. Handover Support

When a mobile node disconnects from its broker, it may connect to another one in a process called handover. The handover is a mechanism that may be used by the mobile nodes to try a better connection with another broker when they are experiencing poor connection quality with their current brokers – for example when a large amount of disconnections are happening in short periods of time.

A broker may also suggest to some of its connected mobile nodes a handover to another broker. This handover may be used by the system to promote load balancing between the brokers, for example, when there is some overloaded broker, or when a new broker is added to the core network.

To avoid message loss during a handover, there must be a special service on the core network that detects messages intercepted by the previous mobile node's broker and forwards these messages to the new broker.

4.2.3. Flexible Services in the Fixed Network

Our model is extensible, in the sense that more nodes can be added to the fixed core network to implement additional services. It should be possible to add these services on the fly (while the application is executing) and to remove them anytime, without having any impact on the other core network nodes. Hence, the core network may feature also other types of nodes executing specific communication, directory or processing services, and not only brokers.

This extensibility may be used to add new services to the system that were not a priori defined. These extra services may be generic (for any application) or to satisfy specific applications requirements. For example, reliability of message delivery can be implemented by a node that stores all recent exchanged messages in a database and listens for notifications from the broker about every node connection and disconnection. This service could then identify when a message was sent to the wrong broker (e.g., the mobile node switched the broker), and can forwarding the message to the correct broker.

4.2.4. Data Production Management

The mobile nodes can exchange messages with any kind of structured data, but since our model is focused at context information in mobile systems, it shall reflect the many characteristics required for handling the dissemination and processing of this sort of information. As mentioned in Section 4.1, Data Providers generate an information object at every Data Update, and this object is pasted into a message that are delivered to all interested Data Consumers. In our model we assume that any mobile node may execute a collection of Data Providers, one for each available embedded sensor, together with Data Providers responsible for producing derived/composed information. The model further establishes that a Data Provider is only deployed and activated when some mobile node requests the corresponding information. The Data Provider is the central concept in our communication model for structured data sharing; Figure 5 illustrated some mobile nodes with Data Providers deployed locally and sharing information among them.

The frequency at which a Data Provider produces Data Updates usually is determined by the type of the information produced, but we also consider some use cases where it is required to configure a Data Provider with a specific Data Update frequency. For example, for HVAC applications is not necessary to collect the temperature of a node's environment every few seconds, but instead, every 15 or 30 minutes, may be plenty enough. However, if the node's sensor is reading the temperature of an industrial oven, it might be necessary to probe the temperature every second, because an increase of the oven's temperature might cause an emergency situation. So, even if the sensor and Data Provider used are the same, they may have different Data Update frequency requirements depending on specific applications needs.

4.2.5. Information Volatility

Context information is usually highly relevant at the moment it was produced and only remains so for a limited time after that, which is often called information volatility. For example, in our industrial oven example from the previous section, it is important to know the current temperature as well as the values from the recent past, e.g. one minute ago, while the oven's temperature from one hour ago is very likely deemed irrelevant.. This requirement of timeliness is one of the reasons why the communication model for mobile nodes should provide on-time information dissemination, so that the information reaches all interested mobile nodes with the least possible delay.

If the application requires needs the history of all Data Updates, this may still be implemented by a special service, that persists all updates for the desired period of time.

4.2.6. Discovery and Deployment of Data Providers

The model should use the same communication network used for disseminating Data Updates also for the discovery of Data Providers. All nodes executing Data Providers must wait for requests of other nodes to produce information. When such a request occurs, the Data Provider responsible for this information is activated and starts to produce the corresponding Data Updates. For the application (i.e., its Data Consumer) it is irrelevant to know the identity or where the Data Provider is executing, instead, it just needs to receive the Data Updates. This is the same characteristic of space and reference decoupling found in the publish-subscribe. If a node needs to locally deploy a Data Provider to acquire a information about the device itself – for example, its current location – this node can request a copy of this Data Provider from any other node that has the required Data Provider on its local repository. Figure 5 showed an example of a copy of a Data Provider from one node to another for local deployment (represented by a dashed arrow in that figure).

Data Providers may be implemented and added anytime on the network, in a similar extensibility and flexibility as the addition of special nodes – i.e., a new application running on any mobile node and providing a newly implemented Data Provider is ready to disseminate Data Updates to any existing applications; moreover, the same Data Provider can be copied for local deployment at any other mobile node.

4.2.7.
Information Hierarchy and Data Updates Flow

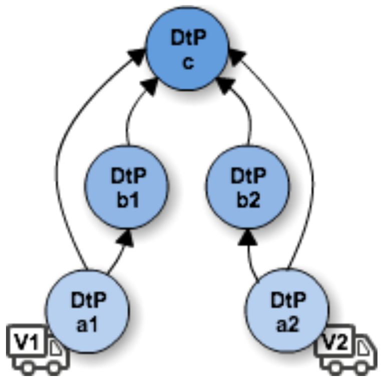


Figure 7 – Information hierarchical transformation example.

To produce derived information, a Data Provider must request the primitive information from the providers capable of producing it. This **dependency relation** generates a tree of connections between Data Providers, where the Data Providers on the tree's leafs produces primitive information that are combined on the parents nodes of the leafs, producing more specialized higher-level information as the level of the tree decreases; Figure 7 illustrate this process. In the picture, the **DtP-a1**, on the leaf of the tree is collecting the location of the vehicle **V1**; based on the information produced over time by **DtP-a1**, the **DtPa2** is calculating and producing **V1** speed. The same process occurs with **DtP-b1** and **DtP-b2**, calculating **V2** speed. **DtP-b1** and **DtP-b2** are instances of the same Data Providers **DtP-a1** and **DtP-a2**, but deployed in **V2**. Combing the information produced by **DtP-a1**, **DtP-a2**, **DtP-b1** and **DtP-b2** it is possible to **DtP-c** to calculate the relative speed that the vehicles are moving from/to each other. Another example is illustrated on Figure 8, where we add **DtP-a3**, responsible for store user personal data, e.g., the user's home location; and **DtP-t**, a Data Provider not related to a unique user or node, but responsible to provide traffic conditions to interested nodes. By combining all Data Providers' information, it is possible to built **DtP-a4**, which calculates the estimated time of arrival of vehicle **V1** at home, considering its current location, speed and the traffic conditions on its route. So, every Data Update produced in the child node generates a notification on the parent node, causing this node to produce a new Data Update itself, notifying its parent and so on. In our model these Data Updates may be produced in high frequency, creating a continuous flow of context information being produced in a bottom-up direction on the dependency tree. It is worthwhile to mention that the relations between Data Providers can be both local (i.e., at the same mobile node) and remote (i.e., receiving information from another mobile node).

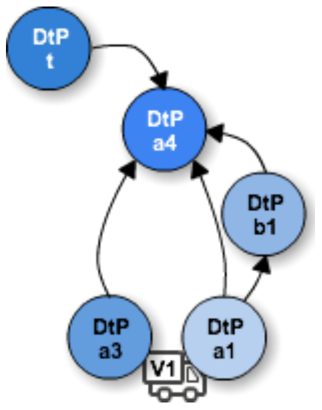


Figure 8 – Information hierarchical transformation second example.

4.2.8. Multiple Communication Modes

Another functionality in the model is the possibility to send arbitrary messages to the mobile nodes. The two basic communication types for the messages are: unicast messages to a specific node or broadcast messages to all nodes. Any node within the core network, any application node in the core network, or by any mobile node can send messages. The model is designed to be a robust, high-performance message dissemination platform even under high load periodic Data Updates messages from all mobile nodes. The main goal is to offer a scalable large-scale communication infrastructure for the development of mobile applications.

Additionally, the model defines that the mobile nodes could be grouped in any set of nodes by any common characteristic, which makes *groupcast* communication possible. Special nodes must implement some group-definition logic and disseminate the group participants to all nodes. For example, a special node on the core network could receive Data Updates about the nodes location and create a group to put together nodes in the same city. Further details will be explained in Chapter 5.

4.3. Mobile-related Requirements

A large-scale communication model for mobile nodes should overcome many limitations, such as: weak, intermittent and unpredictable connectivity, low throughput and larger transmission delays, for example. In addition, it must supports the heterogeneity regarding the mobile nodes' resources and software platforms. All these characteristics require special capabilities of the model. In this section we discuss each of these special capabilities that we believe to be essential for the implementation of reliable and efficient communication between thousands of mobile nodes.

4.3.1. Reliable Mobile Communication

Despite the wide adoption and use of cellular/mobile networks (2G, 3G, LTE), these networks still experience moderate and variable quality of service, which not rarely causes temporary disconnections of the mobile devices' data connections. A disconnection is undesirable for any application that needs to maintain an open connection with any other mobile node. If we use TCP connections, on every disconnection at network layer 2 (L2) all these connections would break and the node would need to restart the TCP handshaking for reconnecting and resuming of data transfers. Additionally, disconnections also affect connectionless protocols, like UDP, because each time the communication with the underlying network is re-established, usually a new IP address is assigned to the mobile device. Hence, if a node switched its IP address, this would require peer nodes to be informed about the new address and managing the resuming of all data already transferred to the previous (now wrong) address. This happens because the assignment of a new IP address is not instantaneous and may cause some disruption of the data flow of packets.

Moreover, mobile disconnections are not only caused by cellular network quality of service. There exist other situations that may cause disconnections that are intrinsic of the cellular connectivity technology. For example, even if a mobile user is served by the best possible connection quality, there always exists places where cellular network connectivity is not guaranteed, such as inside an elevator, in an underground station, or other insulated places. So, because mobile connections are inherently weak connections and, on the other hand, several applications need reliable mobile communications to/from the mobile devices, it is paramount to have a communication model that compensates the problems of intermittent connectivity.

Additionally, modern mobile devices are usually equipped with more than one network interface and may switch the network interface during their use. For example, if a mobile device connected via 3G enters a place where it had previously used a Wi-Fi connection, it might automatically switch from the cellular connection to the Wi-Fi connection, since the latter is probably cheaper, faster and more reliable. This switching is called vertical handover, and in this process the device will be assigned also a new IP address in a different network domain. So, the disruption caused by a vertical handover is similar to the problems of intermittent connectivity, and both should be handled in a similar and application-transparent approach.

So, regarding the problems of mobile connectivity, we need a communication model that is resilient to temporary disconnections and IP changes. In our model, a connection is only considered broken if the mobile node is not able to re-establish connectivity within a threshold period of time. Furthermore, after each of these reconnections, all data dispatched for transmission by the application during the offline period (outbound or inbound) must be automatically buffered and retransmitted. Moreover, the model should define a concept of reliable virtual connection that automatically handles IP address changes of a node and maintains the communication seamlessly as any disruptions happened.

Thus, a communication platform aiming the management of thousands of reliable mobile connections needs to consider the frequent disconnections faced in this environment, and must offer efficient support for shielding these applications from the underlying intermittent network connectivity.

4.3.2. Efficient Bandwidth Usage

Due to the limited bandwidth provided by some wireless technologies and cellular services, the communication model should ensure that the mobile connections are used only when it is really necessary to transmit data of interest of the mobile nodes. This way the network channel would not be misused or overused with, for example, routing related protocol messages, which may cause higher transmission latency or even drop of messages when using a weak wireless connection.

Typical reliable network protocols, like TCP, have a large amount of signaling traffic for flow control and to ensure reliable data transmission. These protocols may also involve complex algorithms (e.g., sliding window) that may require some processing at the communicating peer nodes. Thus, our model requires a mobile communication protocol that is optimized and lightweight in the use of the communication channel for protocol-specific control messages and in the use of the node’s resources.

4.3.3. Space Decoupling

Our model assumes that every node in the system can be a potential source of any information. These information producing capabilities are a direct consequence of its embedded sensors, as well as its processing, storage and communication capabilities, and are independent of the node's identification, type, or location. For example, a node equipped with a thermostat sensor may announce the provision of the *temperature* context information, and should be contacted by any other node to provide the current temperature in the region it is located. By such space decoupling, any other node may request (or subscribe to) any information produced by a node, without having to know which node is specifically producing the information. This space decoupling can be achieved by a publish-subscribe communication paradigm described in Chapter 2.

4.3.4. Unique Node Identification and Unicast Communication

In spite of the space-decoupling requirement, every single node in the system must be individually identified and addressed. This makes it possible to support also private unicast communication between any two mobile nodes, which is also required by some applications.

4.4. Discussion

The model described presents some benefits for mobile applications that share data between the nodes. Comparing with the related work described in Chapter 3, our model aims to provide reliable communication with mobile nodes, reliable and fast message and an information transformation, composition and dissemination infrastructure that can continuously spread the produced information.

Regarding the mobile communication, the model can suppress the deficiencies faced in cellular networks and provide a mechanism that offers reliable communication even in intermittent connections. A mobile node can become disconnected from a threshold period and still maintain a virtual open connection and do not lose any messages. Even if the mobile node gets a new IP address its connection will not break.

The handovers brings flexibility to the mobile nodes to choose the broker that provides better connection quality from its network. The handover is also a mechanism that can be used by the model itself to promote load balancing regarding the number of mobile nodes connected in each broker.

Since the Data Providers are loaded and started on demand, the model indicates a responsible use of resources on the mobile nodes. Data Providers will only be activated and produce Data Updates when at least one node is interested in the associated information. This premise may avoid the waste of resources to produce information that is not being used by any node. Since the communication network connects all mobile nodes, it is possible to have these Data Providers running at only one mobile node and sharing the produced information to any other node using it. For example, if some nodes are interested in the weather conditions at a part of the city, it is necessary that only one node at this location uses it temperature sensor to acquire the temperature and share it, through the network, with all interested nodes.

Using a fixed network and brokers as proxies also implicates in better resources usage at the mobile nodes. For example, if Data Provider deployed in a mobile node is sending Data Updates to hundred of interested nodes, only one Data Update must be sent to the broker which will replicate the Data Update to all interested node using the fixed network.

The possibility to configure the Data Providers with a specific frequency for the Data Updates is an important flexibility not only because it can save mobile node resources but also because it makes possible to build applications with different requirements about updated information. For example, a Data Provider that probes a thermometer can be used differently accordingly with the application use of this Data Provider: if it is a thermometer placed outside on the environment, the application probably don’t need very frequent Data Updates; however, it the thermometer is on a microwave, it will likely be important to know the temperature every couple seconds.

Finally, the context information hierarchy tree is a powerful and flexible mechanism to produce and disseminate high-level information by combining, processing and transform any number of lower-level data. Since it is not a fixed tree, new Data Providers may be implemented and deployed seamlessly on-the-fly, and newly added Data Providers can also be used to produce even higher-level information. Any new Data Update will trigger Data Updates from all Data Providers in the higher-levels of the tree, producing a flow of updates – as was illustrated in Figure 7. These dependencies guarantee that each Data Provider reflects the actual state of its specific information in respect to the whole contextual state. The dependency tree values represent a snapshot of the universal context.

4.5. Assumptions

Even while trying to specify a flexible and resilient model, some assumptions must be made to guarantee its desired behavior. Despite the focus on mobile nodes, which has limitations, we assume that the nodes on the core network are nodes with high availability and with high network throughput, likely servers machines. This way it is easier to guarantee a minimum quality of service on the communication between the mobile nodes.

The model is resilient to the mobile nodes cellular connection intermittency, however to maintain the message delivery reliability the disconnection time must not extrapolate a previous defined threshold. If the threshold is exceeded the mobile node connection with its GW must be considered broken and the buffered messages to and from the mobile nodes considered lost.

To ease the implementation and to avoid undesired behavior, the maximum number of concurrent mobile nodes connection on each broker must be defined, so the system performance won't degrade by a huge number of connections using all the broker's machine resources. If a higher number of mobile node connections are needed, more brokers instances must be deployed to meet the demand.

Finally, to avoid starvation on the applications, there must be guaranteed that all requested information has a deployable Data Provider available on the network, so no request will be unsatisfied. Similarly, all the applications know previously which information is available for utilization.

5
Implementation

The scenario presented in Chapter 1 and the limitations of the current related work presented in chapters 2 and 3 to offer a suitable solution to such scenario, specially the mobile-related DDS limitations, motivated us to design and implement a middleware that extends DDS' high-performance communication capabilities to wireless-connected mobile devices. We try to ensure QoS-like guarantees to the mobile devices' communication, specifically delivery reliability and low latency dissemination of messages.

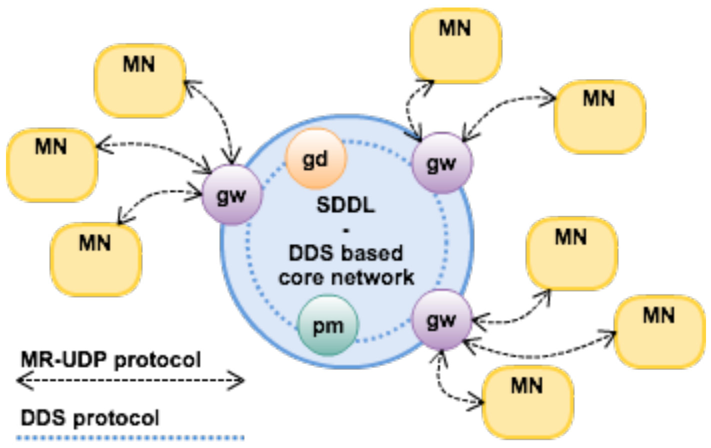


Figure 9 – Model implementation overview.

An overview of our model is illustrated in Figure 9, which shows a set of mobile nodes (yellow squares) connected to another set of fixed nodes (*gw*, *gd*, and *pm* colored circles). The set of fixed nodes is forms the **core network** and is composed of nodes that have stable and good connectivity, abundant computational and memory resources, high availability and reliability, and virtually no energy constrains. All nodes in the core network must interconnect through a high performance wired network with quality of service guarantees regarding message delivery reliability, high throughput and low transmission latency, such as a Local Area Network or a dedicated switching network – in our current implementation we use DDS. On the end of the spectrum, the mobile nodes constitute a set of heterogeneous devices connected through a less reliable IP-based cellular or wireless network, and with stringent limitations in regard to processing, storage and energy resources. Thus, the mobile nodes must use a lightweight communication protocol – with reliable message delivery – to connect with the nodes on the core, so as to guarantee that no data transferred to/from a mobile node is lost. For this purpose, we implemented the Mobile Reliable UDP protocol (MR-UDP) protocol, which will be further detailed in the following sections.

The work detailed in this chapter (and in the whole thesis) is part of a larger project called ContextNet [24], aimed at developing a middleware for on-time communication, coordination and collaboration in large-scale distributed mobile applications composed by thousands of nodes. In the scope of this project, the middleware presented here shows the basic layer for communication and the information/data management parts. The Scalable Data Distribution Layer (SDDL) is a communication middleware that connects stationary DDS nodes in a wired “core” network to mobile nodes with an IP-based wireless data connection. Likewise, the Structured Data Dissemination Service (SDDS) is a middleware layer that offers data management features that implements our Data Sharing Network defined in the previous chapter. The SDDS will also be explained further in this chapter.

The Scalable Data Distribution Layer (SDDL) employs two communication protocols: DDS’s Real-Time Publish-Subscribe Protocol RTPS [53] for the wired communication within the SDDL core network, and MR-UDP for the inbound and outbound communication between the core network and the mobile nodes. The core elements rely on DDS Data Centric Model, where we defined some DDS Topics to be used for communication and coordination between these core nodes, which will be further detailed.

5.1. Mobile Connection Protocol – MR-UDP

The mobile connection protocol is a very significant part of the SDDL, since the whole model (specified in Chapter 4) aims to support efficient and scalable communication with mobile nodes. The previously cited mobile network limitations should not interfere with the system performance and reliability. Instead, we ought to specify and implement a solution that hides all the weakness of the mobile communication from the application developers and provide a reliable and fast message delivery service.

The first solution one would think is to use a TCP based protocol, since it is a reliable protocol. However, when used for mobile communication the TCP is not an adequate solution, since its implementation and details could use a large amount of the devices resources, and we also want a protocol light in resources usage. Another undesired behaviour in TCP is that when a connection is broken – a situation that is frequent in mobile communications – the ongoing communication is also broken and a new connection must be made and all data exchanged again.

So, one possible solution is to use an UDP based protocol, because it is very light on resources consumption and has a simple implementation with no high-level control over the transferred data. However, UDP has no reliability at all, no confirmation on any technique that guarantees that the transferred data is received. Thus, we need a high-level protocol on top of UDP that offers message delivery reliability.

There are several freely available *Reliable UDP* (R-UDP) specifications and implementations. We chose an implementation by Adrian Granados[6] (which is based on the draft available in [56]), because it's an open source Java implementation, which can run on any platform with a JVM implementation, including Android. The fact of being open source was important because we knew that intermittent connectivity resilience is absent in most R-UDP implementations, and thus would have to be added to the protocol.

Hence, we implemented a protocol called Mobile Reliable UDP (MR-UDP) [57], which is a reliable UDP extension with focus on mobile communication. Compared with the original R-UDP, MR-UDP includes mobility-oriented optimizations and extensions in following aspects: transparent continuation of a MR-UDP connection across IP address or port changes; small number of connection maintenance packets for Firewall/NAT traversal; reduced use of mobile device resources and flexible use of threads.

All these optimizations are important for wireless connectivity. For example, when a mobile node (MN) enters an area with no – or weak – connectivity, it may suffer a temporary disconnection and when the wireless signal is back, the node may get a new IP address (via DHCP). In this situation, whenever the disconnection time is shorter than a threshold (e.g., 2 minutes), MR-UDP will keep the original logical connection and all buffered UDP packets will be delivered in the original order.

6 **Simple Reliable UDP – http://sourceforge.net/projects/rudp/**

5.1.1. Main Characteristics

MR-UDP is implemented in Java. The programming interface provides *ReliableClientSocket* and *ReliableServerSocket* classes, which extend the conventional Java Socket's programming interface, with the well-known *Socket*, *ServerSocket*, *InputStream*, and *OutputStream* classes. MR-UDP has several features inherited from Reliable UDP, such as: acknowledgment of received data packets, in-order packet delivery with selective retransmission of lost packets, and over-buffering. In addition, it has the following singular characteristics that are described in the following sections.

5.1.1.1. Simple Flow Control

A *ReliableClientSocket* can be used to transfer large amounts of data, by invoking the *send()* primitive at high frequency (e.g., every few milliseconds). At the other end of the connection, the *ReliableServerSocket* may handle thousands of simultaneously MR-UDP connections with remote peers executing the *ReliableClientSocket*. Since the underlying protocol is UDP, it does not provide any flow control or delivery guaranties. On the other hand, Reliable UDP implements only a best effort reliable delivery of packets, and hence operating system buffers may suffer overflow when large chunks of data are transmitted frequently, causing a progressive and silent (without warnings) loss of packets. So, in order to avoid that the operating system socket drops packets while exposed to bursts of data packet reception, MR-UDP defines a small, configurable waiting time between consecutive sends, at the sender side. While this flow control is much more simple that TCPs sliding window protocol, it is quite effective in most cases (i.e., using 1ms inter-send wait time), and does not require to keep any state of the logic connection.

5.1.1.2. Use of UUID

Each node has a unique identifier (UUID) that is generated only once, when the MR-UDP *ReliableClientSocket* or *ReliableServerSocket* object is created for the first time at a given mobile node. This 128 bits identifier is used by the *ReliableServerSocket* to identify the logical connection with each MN, independently of the IP address and port number being currently used by this MN. Thus, the mobile node may switch networks and receive a new IP address, but will still be recognized as the same MN by the *ReliableServerSocket*. The

UUID also enables to resynchronize the state of the communication (by retransmitting buffered packets which were not confirmed) on both connection endpoints. To implement the unique MN identification, MR-UDP maintains an up-to-date map that records the association between each UUID and its current IP address and port, expressed as the function MN-UUID → "MN-IPAddress:Port". However, the creation and use of a UUID is not mandatory: i.e., if not used, the MR-UDP protocol will work as the original R-UDP, without being resilient to IP address and port changes.

5.1.1.3. Types of Protocol Messages

MR-UDP implements the following types of messages (a.k.a. segments) with the corresponding functions:

- **UID Segment** – carries the UUID of the sending node. When using the UUID identification, this segment is periodically sent, which makes it works also as a heartbeat control message to keep a connection open to a peer node behind a firewall/NAT (this will be explained further in this chapter). This segment is an addition of MR-UDP as an extension of the original R-UDP protocol;
- **SYN Segment** – is used to initiate a new - or reset an existing - connection and to synchronize the packet sequence numbers. It also contains negotiable parameters and optional flags;
- **DAT Segment** – carries a data packet;
- **ACK Segment** – is used to acknowledge in-order received packets. It has an ACK sequence number and contains also the sequence number of the next expected data packet;
- **EAK Segment** - is used to acknowledge out-of-order received packets. It carries a list of sequence numbers of the received packets;
- **RST Segment** – is used to reset the connection by closing and reopening it. When received, a peer node must not schedule any new packet for transmissions, but only try to deliver the not yet acknowledged packets;
- **FIN Segment** – is used to close a connection;
- **NUL Segment** – in the original R-UDP protocol this message is used to check if the node at the other end of the connection is still alive (i.e., AreYouAlive? Message). When received, and if the connection is still active, the peer node must immediately acknowledge it. In MR-UDP,

this segment is used only if a MN does not use UUID for its identification.

5.1.1.4. Retransmission of undelivered messages

MR-UDP maintains a buffer of all “not yet acknowledged” packets for each connection. For all packets in this buffer, it tries to re-send them a certain number of times, and wait for the corresponding acknowledgement. When the configured threshold of retries is reached, the node considers that the connection has been dropped, and sends a *FINSegment*.

5.1.1.5. Data segmentation

Large messages are split into blocks of data, each of which is sent in a separate *DATSegment*. The block size is configurable, but in experiments we noticed that block size of 384KB is a good choice for the mobile network scenario. It minimizes the chances of dropping packets (due to overload the networking component of the operating system), and the costs of packet retransmission, while keeping the overhead of acknowledgements manageable and providing a good transmission bandwidth despite use of the inter-send wait time, MR-UDP’s simple flow control mechanism. The total size of the MR-UDP buffer is also configurable by a desired number of blocks that must be retained.

5.1.1.6. Connection Management: Disconnection Detection and Firewall/NAT Traversal

Any endpoint of a MR-UDP connection detects a disconnection when: (a) it does not receive any segments from the remote endpoint for some time interval; (b) it sends a *NULSegment* or *UIDSegment* to the remote endpoint and does not receive an *ACKSegment* back within some time interval (c) it has reached the re-transmission threshold for non-acknowledged sent packets (in the connection’s buffer).

When using the UUID identification some disconnections that would happen can be hidden for the application that uses MR-UDP. When a MN experiences a temporary disconnection from its network and gets a new IP address upon the reconnection, it immediately sends a *UIDSegment* with its new address. This enables the *ReliableServerSocket* to learn the node’s new location and update the UUID-IPAddress:Port map. Thus, in this situation the conventional Reliable UDP connection would be broken, but the MR-UDP continues to work as if the connection were never lost.

Also by using the *UIDSegment* (or the original *NULSegment*), a MN behind a Firewall/NAT can keep a connection active, and allows other nodes to reach it. This is because the network routers will maintain a table with the information to reach the packet sender back in the opposite direction. In some sense, this mechanism replaces piggybacking of near real-time messages or control information, as used in other approaches, and makes possible to reach the MN without the need to wait for a message/signal originated from it. To traverse Firewalls, it is necessary that the mobile node initiate the connection, which is the case in our model.

5.1.2. MR-UDP Overview

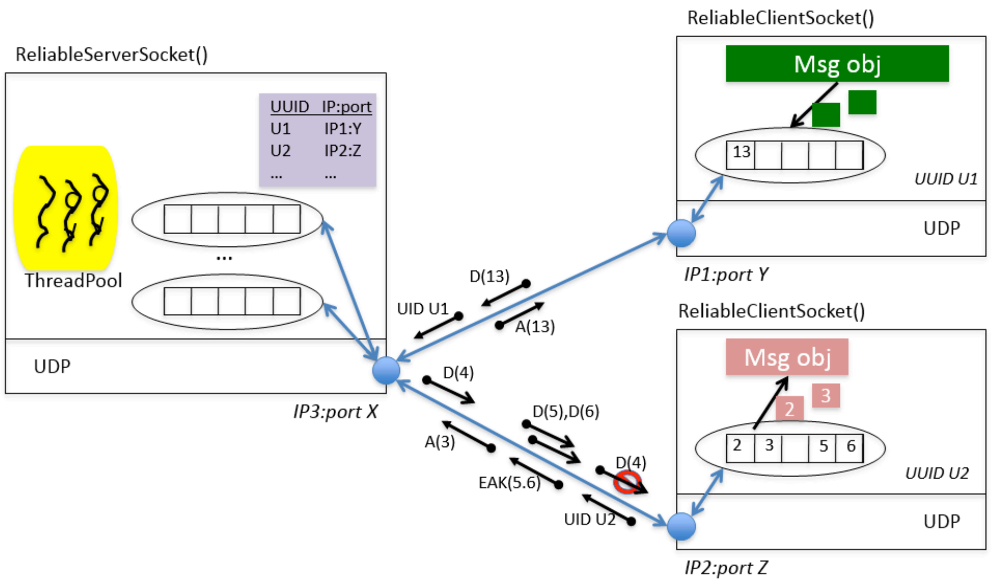


Figure 10 – Schematic view of MR-UDP in action with two clients.

Figure 10 illustrates some of MR-UDP's features and behavior. On the left side, a *ReliableServerSocket* is handling several connections (ovals) with remote clients by multiplexing them over its local port X. A pool of threads does this handling. More specifically, it shows two connections – and the associated packet buffers – for clients with UUDIs U1 and U2. Notice that a hashTable, the UUID-IPAddress:Port map (purple box) associates each client to its current Internet address and port. If some of this information changes for any known client/UUID, for example, if the corresponding MN is connected to a new network, then the *ReliableServerSocket* just updates this map. On the right side, client U1 is serializing and segmenting a message object for transmission into packets and putting these packets, one by one, into the OutBuffer. After data packet with sequence number 13, D(13), arrives at the *ReliableServerSocket*, a corresponding acknowledge packet, A(13), is sent and received by the *ReliableClientSocket*. Client U2, on the other hand, is reassembling a message object – to be delivered to the application – from its constituent packets D(2) through D(6). After acknowledging packet with sequence number 3, data packet D(4) is lost, but D(5) and D(6) arrive. Client U2 then sends an EAKSegment, EAK(5,6) packet, which causes the *ReliableServerSocket* to re-transmit data packet D(4). In the meantime, U2 sends another UIDSegment to keep the connection through a Firewall open. The Figure also suggests that MR-UDP just uses a single port (blue dot), associated with the UDP socket.

5.1.3. Additional Implementation Details

Since we wanted the MR-UDP protocol to efficiently handle communication in mobile networks, it was necessary to implement several optimizations and extensions to the original R-UDP.

The first one that deserves notice is the use of a pool of threads. In MR-UDP, the pool of threads has small, configurable number of worker threads, which are automatically increased as new connections are being handled (by the *ReliableServerSocket*). By this, it is able to manage thousands of simultaneous connections. In the original R-UDP implementation that we used as the starting point of our implementation, every connection used 15 threads! But since threads consume much system resources (creating too many threads in one JVM can cause the system to run out of memory or thrash due to excessive memory consumption), this implementation was not scalable and the result was that many connections were dropped. As all threads in the original R-UDP were used for timeouts (in a class called *Timer*), in MR-UDP we just implemented a more efficient Timer using the pool of threads. Our tests have shown that with just 3 worker threads, MR-UDP incurs in much less resource consumption, and is able to handle 5-10 times more connections. Moreover, a reduced number of threads together with the simple flow control and adequate configuration of the protocol message frequencies helps to implicitly reduce energy consumption.

When using a transport protocol over an unreliable network, it is important to enable that higher-level protocol layers be notified of some states of message delivery or connectivity. For this reason, in MR-UDP we made some changes to the message send and acknowledge APIs. We used the Observer design pattern that enables high-level protocol layers to observe and be notified when specific events happen within MR-UDP. For example, we made MR-UDP inform which was the ACK number of a message sent, and enriched the internal socket listener to inform the number of every ACK received. In this manner, the software layer above MR-UDP can be informed if a message was correctly delivered at its destination, or if a message was not delivered due to disconnection.

To keep current applications compatible, we made *ReliableClientSocket* (and *ReliableServerSocket*) extend the default Java socket classes, which means, that any application can use these sockets without significant modifications. Basically, MR-UDP's reliable socket will be created using another constructor (that will inform the UUID), and nothing else needs to be changed.

Another noteworthy characteristic is that MR-UDP implements the Front Controller design pattern, allowing the *ReliableServerSocket* to use a single UDP port for all connections, i.e., through which all inbound and outbound message traffic is multiplexed/de-multiplexed. For this, as already mentioned, MR-UDP maintains an up-to-date map of the MN's UUID and its current pair (IPAddress: Port).

Finally, we also did some re-factoring to make MR-UDP Android-compatible, since some Java constructions were causing problems in some cases (e.g., causing deadlocks) when running MR-UDP in Android's Dalvik JVM.

5.2. Core Network

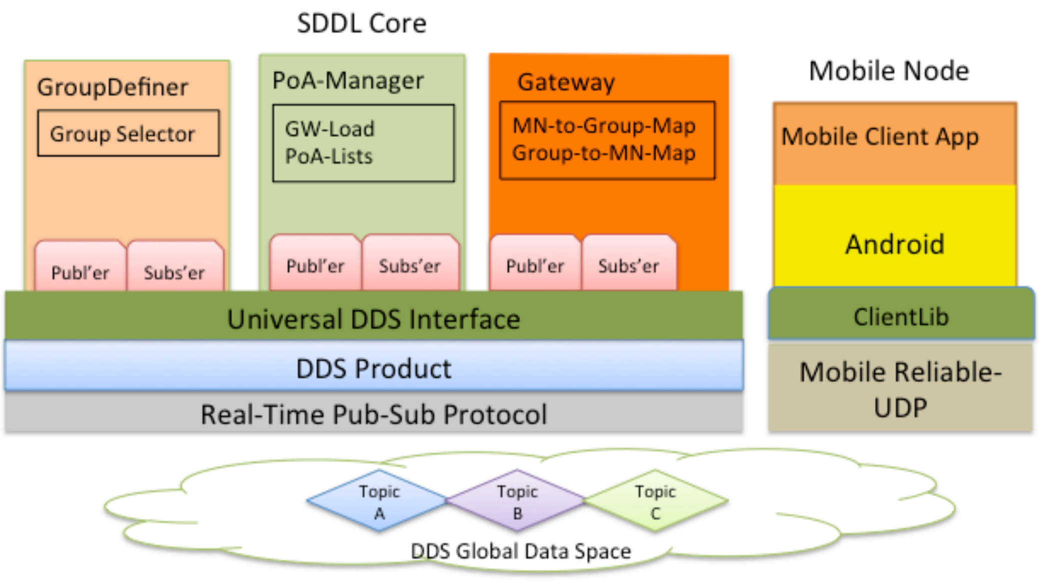


Figure 11 – Nodes and protocols used in SDDL.

The core network uses DDS as the communication substrate to connect stationary nodes. As part of the core network, there are three types of SDDL nodes with distinguished roles that will be explained in the following sections and are depicted in Figure 11.

5.2.1. Gateway

The Gateway (GW – see Figure 9 on page 49) defines a unique *Point of Attachment* (PoA), for connections with the mobile nodes, it is the implementation element we used for the broker defined in our model in Chapter 4. The Gateway is thus responsible for managing a separate MR-UDP connection with each of these nodes, forwarding any application-specific message or context information into the core network, and in the opposite direction, converting DDS messages to MR-UDP messages and delivering them reliably to the corresponding mobile node(s). The GW is also responsible for notifying other SDDL core network nodes when a new MN becomes available, or when some MNs disconnect from it. This information is necessary to implement other SDDL services in other core nodes, e.g., a service that caches messages addressed to temporary offline mobile nodes, for posterior delivery.

Communication and coordination between the GWs is done by special DDS Topics created for that purpose. There are two main types of topics. The *SystemTopics* are the topics used for internal system and services coordination and are hidden from the applications. For example, there is a *PingTopic*, which can be sent from any GW and instruct all the nodes in the core network to respond to the topic; this topic can be used to check the overall network performance. Special services can also implement their own SystemTopics to be able to communicate among the nodes participating in their services providing. For example, a distributed service for logging can create a topic to carry the data that all nodes must persist. The *ApplicationTopic* is the topic used to carry data from/to applications that use SDDL. For example, when a node sends its location or a private message to another node, the GW translates this message and publishes the data in an ApplicationTopic to all interested nodes. In addition to the data, the topic has many additional metadata about the publication, e.g., sender identification and/or receiver identification. The GW uses these extra data to be able to perform filtering at the subscription of the topic. For example, the GW should only subscribe to private messages sent to mobile nodes connected to it.

5.2.2. PoA-Manager

The PoA-Manager (*pm* in Figure 9 on page 49) is responsible for two things: (a) to periodically distribute a *Points of Attachments List* (PoA-List) to the MNs, and (b) to eventually request some MNs to switch to a new Gateway/PoA. The PoA-List is always a subset of all available Gateways in SDDL. The order in the list is relevant, i.e., the first element points to the preferred Gateway/PoA, and so forth. By having an updated PoA-List, a MN could switch its Gateway if it detects a weak connection or a disconnection with the current Gateway. Moreover, by distributing different PoA-Lists to different groups of mobile nodes, the PoA-Manager is able to balance the load among the Gateways, as well as announce to the mobile nodes when a new Gateway is added to the network, or an existing Gateway is removed (or failed) from the SDDL core.

5.2.3. GroupDefiner

GroupDefiners (*gd* in Figure 9 on page 49) are responsible for evaluating group-memberships of all mobile nodes. To do so, they subscribe to the DDS topic where the Data Updates are disseminated, i.e., the ones sent from the MNs and forwarded by the GWs, and map each node to one or more groups, according to some application-specific group membership processing logic. This group membership information is then disseminated to all Gateways in the SDDL core network, using a SystemTopic. The GW is responsible to maintain a structure about group-membership information for every MN connected to it. Whenever a new message is sent to a group, each Gateway queries its group-to-MN mapping, to learn to which of the connected MNs it must send the message. The current groups of a node can be determined, for example, by its node ID, its current position (e.g., if it is inside some region), or by any other attribute/field of its context information (e.g., a node’s battery level). In any case, it is important to notice that logic to define the groups is always application-specific and should implemented by the application developer and deployed in the GroupDefiner.

5.2.4. Architecture Overview

Figure 11 shows SDDL nodes types discussed so far and the communication protocols they use. On the mobile side, a mobile application (currently we only support Android) uses a client library (ClientLib) for establishing and managing MR-UDP connections to send and receive messages to a Gateway. On the SDDL core network side, all nodes use a DDS implementation-independent Universal DDS Interface (UDI), whose API classes and methods are mapped to the different primitives for setting up and configuring the communication entities of each DDS product, which in turn uses the DDS standard high-performance RTPS protocol. The Gateways are the only nodes in the SDDL core that also use the ClientLib to manage (an arbitrary number of) mobile connections. The Controller is a Java Applet that interacts with a JavaScript for displaying all MN's current locations on a map, with a Web browser window. The Web browser also displays the current groups of MNs (which can be defined and managed by the user), and supports operations to send (uni-/group-/broadcast) messages to, as well as receive text messages from the MNs. The other elements of SDDL shown in Figure 11 will be explained in mode detail along the remainder of this chapter. While several people in our lab participate in the development of SDDL/ContextNet, the author of this thesis has contributed in the overall system architecture and participated in the high-level specification of some parts of all components presented here. In terms of actual implementation, the author was responsible for the entire development of the following components: MR-UDP, ClientLib, GroupAPI, PubSubAPI and SDDS.

5.3. Handling Mobile Handover

A *Handover* (HO) happens when a mobile node connected to a Gateway drops or loses its connection and connects to a different Gateway. SDDL supports both *core-initiated handover*, i.e., when a mobile node is requested by the PoA-Manager to connect to a new GW, and a *mobile-initiated handover*, i.e., when the mobile node spontaneously decides to connect to a new GW. In both cases, it is the mobile node that actually chooses another PoA from its PoA-List.

While performing a handover between Gateways, i.e., during the period of time when the node is temporary disconnected, it is possible that some messages fail to be delivery to it. In order to enable reliable delivery of messages during a handover (a.k.a. *smooth handover*), SDDL also implements the *Mobile Temporary Disconnection Service* (MTD), which may run on any node(s) of the SDDL core network.

The MTD is responsible to listen to *disconnected-MN* messages produced by the Gateways, and subsequently collect all messages sent to the mobile node during its HO offline period. As soon the node is connected to a new GW – which will also be announced by a *connected-MN* message – the MTD Service will send all the collected messages to the node through the new GW. Since not all applications require such reliable delivery, the MTD Service is optional in SDDL. And, of course, the buffering capacity of MTS is limited by the amount of memory allocated to it at deployment time. So far, we have not implemented any specific garbage-collection algorithm for minimizing message loss due to buffer overflow.

5.4. Nodes IDs and Message Delivery

In the SDDL, every mobile node and every Gateway has a unique identifier (ID). Every MR-UDP message carry the mobile node’s ID, and the ID of the GW currently serving the mobile node is automatically attached to any message entering the SDDL core network. By this, any corresponding node can learn which is the mobile node’s current GW, and most messages addressed to a mobile node will thus carry also a Gateway ID, allowing them to be directly routed to the corresponding Gateway serving the mobile node. However, if the mobile node becomes suddenly unreachable/disconnected, its most recent Gateway, will notify this to all other nodes in the SDDL core network, and any future message to the node will omit the Gateway ID. However, even in this case where the current Gateway of a mobile node is not specified, messages to the mobile node will be delivered, because they will be received by all Gateways but only properly forwarded by the correct GW. Hence, as soon as any of the Gateways gets a new MR-UDP connection from the ‘lost’ mobile node, it will forward properly forward future messages to the mobile node including the new GW ID.

As SDDL’s basic functionality, it is possible to send two types of messages: *UnicastMessages* to a specific node, or *BroadcastMessages*, to all active nodes. In the SDDL core, sending a message is just as simple as writing into the DDS *PrivateMessageTopic* the message payload object, and the ID of the mobile node, or, alternatively, setting a *broadcastFlag*. As mentioned before, it is further possible to set also the Gateway ID, in which case all the Gateways not serving the mobile node will filter and discard the message.

Messages can be sent by an SDDL-specific node within the core network, by an arbitrary application node in the core network or by any mobile node. In the latter case, the mobile node just sends the message payload object, the ID of the addressee (or the broadcastFlag) through its MR-UDP connection and does not care on how the message is delivered to its destination. In out tests, we achieved message delivery times of less then *200ms*, and experienced no message loss when the mobile node is connected.

Applications developed using SDDL can include also *groupcast* communication, whose group-definition logic is processed at the GroupDefiner nodes and the group-membership information is disseminated to all Gateways, by which they are able to update their MN-to-Group and Group-to-MN mappings, and as mentioned before and will be explained in more detail below.

5.5.
Group Communication and Management

Applications developed using SDDL can extend message delivery functionality by implementing a classifying service on a core network node. This service is responsible to classify all mobile nodes into groups, according to any criteria, such as by mobile node ID, current position, its speed, or any other particular application specific context. Once the groups are defined, it is possible to send messages to all nodes of any group. In this case, only the Gateways that have some node in the group will subscribe to group-specific messages and will forward them to the locally served mobile nodes that are group members.

Groups of nodes may be either long-lived/explicit or context-defined. In the former category they are explicitly defined by the application developer/operator, e.g., nodes belonging to a certain user group, to a same company or administrative domain, or nodes of a same type. For context-defined groups, the membership of a node is dynamically determined by its most recently updated context data (via a Data Update – DtU). For example, if the context means the “geographic position”, then all nodes located within a certain region (e.g., a metropolitan area or within the boundaries of a state), can form a context-defined group. Alternatively, nodes could also be grouped by their current type of connectivity (3G vs 2G), their residual energy level, accelerometer data, local weather condition, or any other dynamic context information. Hence, context-defined group membership has to be continuously updated according to the most recent DtU sent by the nodes, and this is done by the GroupDefiners in tandem with the Gateways: for each DtU the GroupDefiners check if some membership changed, and if this is the case, disseminate this node’s group change to all Gateways, which update their mappings accordingly.

Each GroupDefiner internally consists of a generic DtU message processing part, and an application-specific, *Group selection module*. The generic part is responsible for reading DtU messages from the DDS domain, recording the current groups related to the message, and handling the DtU object to the Group Selection module. This module will execute a specific group-mapping algorithm to determine the group/s that the corresponding producer of the DtU is member of and must be implemented accordingly with any application specific rule.

This split between the generic and the specific group membership processing parts has some advantages: (i) it is possible to deploy several GroupDefiners in the SDDL core, each of which executing a Group selection module that examines a certain type of the DtU object independently of the other modules, and (ii) Group selection modules may be easily exchanged in the GroupDefiner, without compromising the remaining function of the SDDL group management and communication capabilities.

5.6. Universal DDS Interface and General Application Topics

The Universal DDS Interface (UDI) is a library that fully abstracts the DDS implementation utilized, promoting reusability and interoperability of SDDL components. The main goal is to hide away the idiosyncrasies of the APIs of each DDS implementation, and simplify the set-up and configuration of DDS entities.

UDI supports the creation of DDS topics (and content filtered topics), Domain Participants, Publishers, Subscribers, Data Readers and Data Writers, as well as the definition of QoS policies for each such entity, all in a straight and uniform way. As mentioned before, the DDS standard defines about 48 possible QoS policies [21, 58], but every DDS implementation has different ways of assigning them to Topics, Domain Participants, Publishers, Subscribers, Data Readers and Data Writers, as well as configuring the corresponding network services, which makes the proper use of QoS of DDS a cumbersome task. Moreover, several DDS products only support DDS setting at build time. Hence, one of our goals in designing UDI was also to simplify this process, and to support QoS setting at deployment time. Therefore, in UDI QoS policies are defined by passing a single QoS policy object at the initialization method, which aggregates the chosen QoS parameter settings for all DDS entities at a single place. Thus, whenever a DDS implementation is to be replaced or added, it requires only implementing the new UDI port to the chosen DDS implementation/product. UDI is also topic independent in that it is able to manipulate any DDS topic, not only the SDDL topics.

As already mentioned, all SDDL core components interact through DDS topics. Some of them are used for control purposes, i.e., for coordination among

GroupDefiners, Gateways, PoA-Manager, etc., while other topics are used for Application messages. Concerning the latter one, SDDL defines a single, and generic Application Topic type, which is to be used by the application programmer to create its application topics. The main components of this topic type are a content attribute, which holds any Java-serialized object, and a list of group IDs for the exchanged message. This single generic topic type makes SDDL a general-purpose communication middleware which is completely agnostic to the application-specific classes, and which is responsible only for the reliable and efficient message delivery to/from the mobile nodes, as well as for the management of group memberships of the nodes.

5.7. ClientLib

The ClientLib is a Java library part of the ContextNet that is provided to applications developers that want to implement applications that uses SDDL as the communication substrate. It hides most details of the underlying protocol utilization (e.g., MR-UDP) and handles several connectivity issues from the communication protocol between the mobile node and the Gateway(s). The current ClientLib implementation was designed to work in Java for desktops and Android.

The fundamental element of the ClientLib API is the *NodeConnection* interface, which provides methods to connect to a remote GW and to send and receive messages. The communication is asynchronous, which means that the applications do not block when calling the *send()* method. There are two *listeners* provided with the API: (a) the *NodeConnectionListener* should be registered before a connection takes place, so the application can be notified when the connection is established or if the connection could not be made – among other notifications; (b) the *SDDLConnectionListener* is provided to applications that want to be informed when SDDL related events occurs, e.g., when a new PoA-List is received. All messages sent by the application to the GW are queued in an internal buffer to be sent as soon as possible. However, if for some reasons some messages could not be send, mostly because a broken connection that could not be re-established, the application will be notified about which messages where not sent in the *NodeConnectionListener* – though it only happens if the node could not make any connection after trying to handover to any of the known Gateways.

Currently, ClientLib has been implemented only for MR-UDP protocol described in the previous sections (implemented in the *MrudpNodeConnection* class). However, the ClientLib implementation is decoupled from the communication protocol that may implement the *NodeConnection* interface, which makes possible to seamlessly integrate other underlying protocols (e.g., HTTP). Thus, with future developments, it could be possible to the application developer to select a protocol that best suits his/her application's needs – since not necessarily all mobile nodes will necessary have stringent resources and network limitations.

The ClientLib also implements and hides from the application developer all low-level SDDL functionalities, like handovers between Gateways, or retransmission attempts for non-acknowledged packets. Concerning handovers, the ClientLib is responsible for handling and managing the PoA-List (i.e., the list of Gateway IP addresses and ports received from the PoA-Manager), reacting to a mandatory handover request, and deciding when to performing a spontaneous handover If the application tries to send any information during the short disconnection period between handovers, the ClientLib buffers the packets and sends them as soon as a new connection is established.

When the client application is running on a mobile node with a very unstable wireless connectivity, with frequent temporary disconnections, the ClientLib tries to shield these disconnection events and reconnection attempts from the application, so that the mobile client application may behave as if the client had a stable and continuous connection. For all communication, ClientLib uses the abstract class *Message*, which is the superclass that all other messages must inherit. An *ApplicationMessage* subclass is already provided for general communication purpose, which must be extended and implemented by the application developer with details and data needed for his application if needed. The application is notified about the receipt of new *Messages* in the *NodeConnectionListener.*

The ClientLib has also a *server* part (*NodeConnectionServer* class), which is used by the Gateways to wait for and handle mobile client connections. The MR-UDP implementation offers a better use of Threads when comparing with the original R-UDP. We have reduced the number of Threads utilized by the protocol from 15 per connection to 5 Threads per 1,000 connections, making possible to the MR-UDP server to maintain several thousands of concurrent connections to mobile nodes. Additionally, the ClientLib is also responsible for the serialization and deserialization of all exchanged data. ClientLib also implements some other features that happen without any interaction with the applications, like the reception and response to *Ping Messages*, which are used by the SDDL to collect statistics about the latency of mobile connections – the application is only notified of these events if an *SDDLConnectionListener* was registered.

5.8. ClientLib's ExtendedAPIs

In general terms, the ClientLib is a high-level protocol API that provides to application developers means of communication among mobile nodes using in the underlying implementation the MR-UDP protocol and the SDDL core network. However, some application developers may not consider the ClientLib API completely suited for their needs, and may want even higher-level abstractions. For this matter, the ClientLib was designed and implemented to be an extensible library where other communication protocols and APIs can be implemented over the basic *NodeConnection* concept – we call them *ExtendedAPIs*. The ClientLib already has two of such higher-level communication protocols and APIs, the GroupAPI and the PubSubAPI (please, see Figure 12); the former provides means for groupcast communication among the mobile nodes and the latter topic-based publish-subscribe communication. Further details will be explained in the following sections.

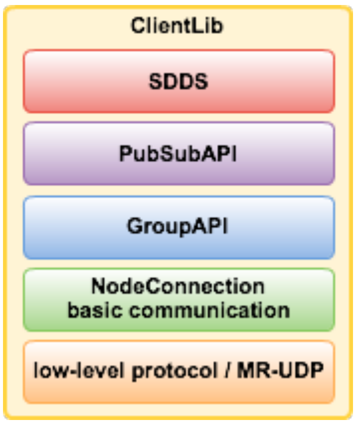


Figure 12 – ClientLib's internal APIs layers.

As Figure 12 suggests, the ClientLib can be used to hide not only the MR-UDP protocol and SDDL functionalities but its *NodeConnection* API as well. If an application is implemented, for example, using the PubSubAPI it will only use its APIs methods and the ClientLib will handle all communication that happens underneath. Moreover, a high-level API can be implemented over another high-level API, which is the case for the PubSubAPI implemented over the GroupAPI.

The ClientLib's extensibility is supported by the class *ExtendedMessage* and the *ExtendedMessageListener*. As explained in the previous section, the basic communication element in ClientLib is the *Message* superclass, which has *ApplicationMessage* as an extension, already provided. However, *ApplicationMessage* has a sibling message type called *ExtendedMessage* (see Figure 13). The ClientLib's implementation regarding the reception of messages hides from the *NodeConnectionListener* any messages types received that is derived (extended) from *ExtendedMessage*. Thus, only messages types extended from *ApplicationMessages* reach the application through the *NodeConnectionListener*. So, to implement a higher-level protocol over ClientLib, the developer only needs to design and implement its protocol messages (extending *ExtendedMessage*) and register an *ExtendedMessageListener* for that type of messages with the *NodeConnection*. To send an *ExtendedMessage*, the developer uses the normal ClientLib's *send()* method. When an *ExtendedMessage* is received from the remote node that the application is communicating with, the *ExtendedMessageListener* is notified. Figure 13 shows an example of the *GroupMessage* type that has two subclasses (*JoinGroupMessage* and *LeaveGroupMessage*), they are used by the GroupAPI implementation (that we will describe in the next section) to implement the group membership management of a mobile node by the application.

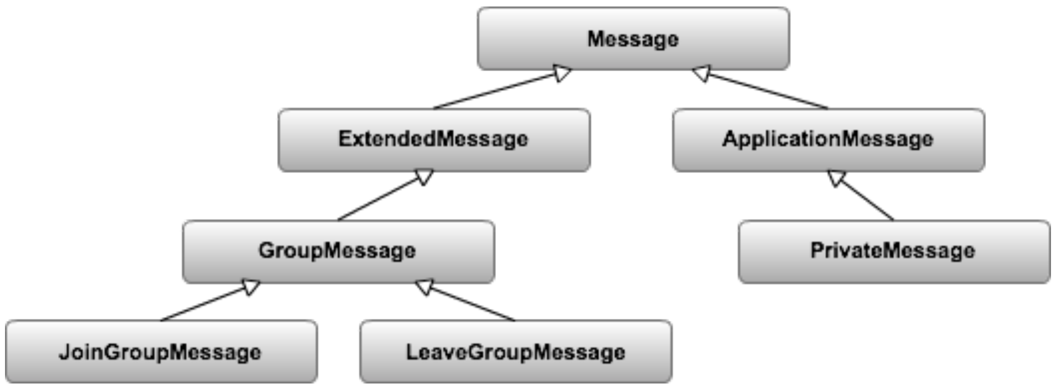


Figure 13 – Message class hierarchy example.

From the point of view of MR-UDP and the SDDL parts, the *ExtendedMessage* is handled as any other message that has some basic fields such as a source and a destination. It is important to note that because of the aforementioned every ClientLib/SDDL feature (e.g., resilience to temporary disconnections, IP address changes and handovers) will work transparently for any higher-level protocol developed on top of ClientLib. This makes it possible to implement new protocols that take advantage of SDDL’s reliability, scalability and on-time dissemination properties – without the need to design and implement them again from scratch.

5.9.
GroupAPI

As its name suggests, the GroupAPI is an ExtendedAPI that provides group communication to applications implemented with the ClientLib. The API implements the *GroupMessage* type that extends *ExtendedMessage* and another few classes that extend *GroupMessages* for the API implementation (for example the *JoinGroupMessage* and *LeaveGroupMessage* types – please see Figure 13). The GroupAPI works in tandem with the Gateway. To create an application that supports group communication, the developer needs to create a *GroupCommunicationManager* and pass a reference to an active *NodeConnection*. Thus, after creation, the application can communicate using exclusively the *GroupCommunicationManager*.

In the Section 5.5 we explained that the SDDL provides means for group communication via groupcast messages that are sent to a specific group of nodes that are managed by the GroupDefiner. A mobile node will either automatically enter one (or more) groups through the GroupDefiner imposition (e.g., which assigns a group to the node based on its context information), or by explicitly calling method *joinGroup()* in the GroupAPI. In both cases the Gateway will add a filter to the DDS messages that carries messages for the groups. The GroupAPI provides a *GroupCommunicationListener* that can be registered by the application to be notified when the GroupDefiner had assigned or unassigned the node to a group and about the incoming groupcast messages of the groups the mobile node is part of.

In SDDL a group of nodes is defined by two integers: one specifies the group ID and the other the group type. It is important to have this pair of identifiers since distinct application could use the same group id for different purposes, so we added a group type for differentiation. Thus, to enter or leave a group, the application must specify the group ID and group type and call the *joinGroup()* or *leaveGroup()* methods in the *GroupCommunicationManager*. The *sendGroupcastMessage()* method sends messages to specific groups and, when called, the GroupAPI automatically encapsulates the message in a *GroupcastMessage* and send it via MR-UDP, SDDL and to therefore reach all mobile nodes participating in that group. Groups are open in the sense that a node do not need to be a member of a group in order to be allowed send a groupcast message to that group. Moreover, the *joinGroup()* and *leaveGroup()* methods only have effect if the node is not already asociated to a group througth the GroupDefiner. If the node is already in a group, it will receive all the groucast messages anyway.

5.10. PubSubAPI

The PubSubAPI is implemented over the GroupAPI and uses it to provide a topic-based publish-subscribe communication service (explained in Section 2.1) to the application. The fundamental element of PubSubAPI is the *PublishSubscribeMananager*, which needs a *GroupCommunicationManagager* and a *NodeConnection* in its constructor.

The application can register publishers of, or subscribe to, publish-subscribe topics. Each topic is defined by a topic name represented by a keyword. The calculated keyword's hash value is an integer used to represent the topic group ID in the PubSubAPI's underlying GroupAPI utilization (e.g., the keyword *"location"* will generate the group ID 74518). The PubSubAPI defines two different ClientLib group types to implement the communication paradigm: one for managing the matching of publishers and subscribers and the other for the event dissemination in each topic (e.g., group types *100* and *101*).

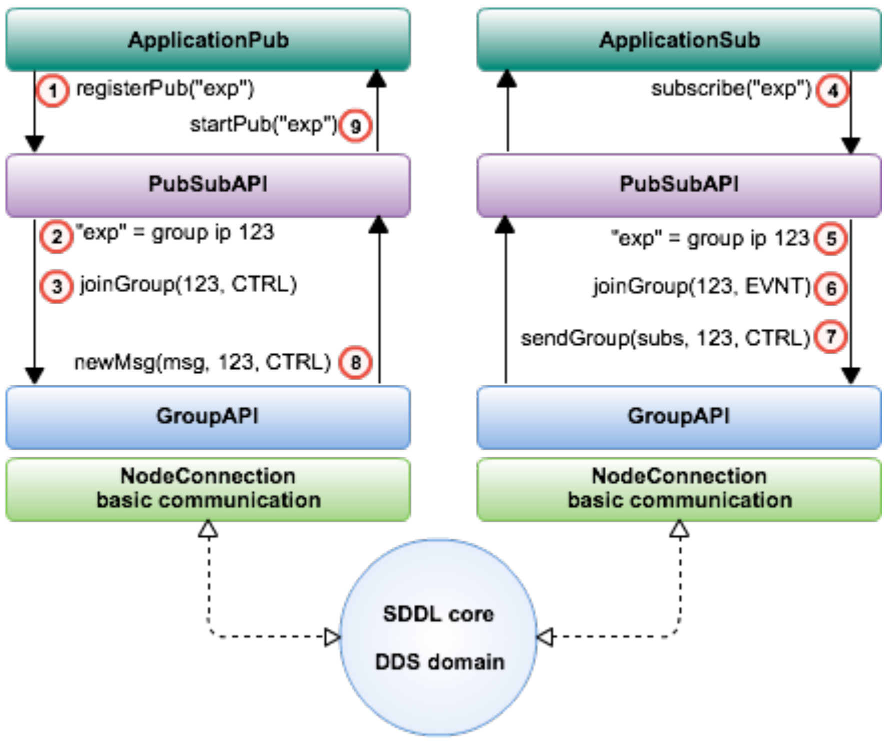


Figure 14 – PubSubAPI example.

Figure 14 shows an example of the PubSubAPI utilization. When an application registers a publisher of some topic (1), the PubSubAPI calculates the topic name keyword's hash value (2) and uses the GroupAPI to join the group ID defined by this integer and the control messages group type (3). When the application is registering a subscriber for some topic (4), the corresponding group ID is also calculated (5) and the GroupAPI is used to join the group ID with the group type where the messages events are disseminated (6). Additionally, when registering a new subscriber of a topic, a groupcast message with the new subscription is sent to the control group type of the topic group (match-making group) (7), which will be received by all mobile nodes that had registered publishers to that topic (8 and 9). At every publication of new events, the publishers send groupcast messages to the event dissemination group type of the topic group ID, which will reach all interested subscribers.

For application development, the *Publisher, Subscriber* and *PublishSubscriberListerner* interfaces are provided and must be used to implement publishers, subscribers and a listener to the PubSubAPI. The *PublishSubscriberListerner* interface is used only if the application needs to be informed about underlying publish-subscribe events, like for example when a match between a publisher and subscriber occurs. A subscription is defined using the *Subscription* class that can optionally attach a *SubscriptionFilter* as a parameter. The subscription filter is implemented by an abstract method that receives the event published and should be implemented by the application developer to return *true* or *false* in the case that the event should be processed by the subscriber or not. Since the filter is attached to the subscription, the publisher has access to all filters in all subscribers of its topic. This way is possible to do the filter processing both at the event source (i.e., the Publisher) and in the event consumer (i.e., the Subscriber). The *Subscriber* interface defined the *newInformationReceived()* method that should be implemented by the application developer and is called by the *PublishSubscribeMananager* when a new event is received.

In general terms, the PubSubAPI makes possible to use all SDDL features using a standard topic-based publish-subscribe API. Applications using it will have the additional benefit of the reliable, scalable, and on-time dissemination of events with the automatic Gateway handovers and load balancing management provided by the PoA-Manager and PoA-Lists. Moreover, since the PubSubAPI has small code footprint, the application can be deployed at mobile devices with stringent resources and intermittent wireless connections.

5.11. Structured Data Dissemination Service – SDDS

The Structured Data Dissemination Service (SDDS) is another extension of the ClientLib. It provides general abstractions and implements a protocol for collecting (probing from mobile sensors), processing and sharing of structured data information among any mobile nodes. The SDDS is implemented on top of the PubSubAPI. It allows the applications developers to consider only the types of information of interest to build its applications, instead of dealing with publish-subscribe topics and their filters. As mentioned before (in Section 4.2.7), SDDS thus supports the implementation and operation of a data sharing application as an arbitrary web of (mobile and distributed) producers, transformers and consumers of different types of information.

Two are the main abstractions used by the applications to interact with SDDS: the Data Provider and the Data Consumer.

The **Data Provider** – or just **DtP** – is the software entity responsible for producing Data Updates of a single information type. Each DtPs is thus specialized in the construction of a ***DataUpdate*** object of single information type. The data is typically obtained by probing a specific sensor embedded in the device, or an external source of information, such as a database, a web service. Through SDDS each ***DataUpdate*** object will be delivered to all the interested Data Consumers, which may be applications or other Data Providers, that may be device local or remote, The Data Providers must be implemented using the *DataProvider* class, indicating which context information they provide and in which platform they can run (i.e., Java, Android or both).

A **Data Consumer** – or just **DtC** – is a subscriber for Data Updates. When an application (or another Data Provider) needs some kind of context data as input, it must provide a *DataConsumer* listener that will be notified by the SDDS when every new Data Update is produced by the DtPs.

In SDDS it is also possible to create Data Providers that subscribe to Data Updates from other DtPs (local or remote), in order to produce aggregate or transformed higher-level information from simpler information – e.g., a high-level DtP that subscribes to the location of a group of mobile nodes that are traveling in a specific road and produces Data Updates about this road traffic conditions. Thus, in this case, the DtP becomes also a DtC that is notified by SDDS of new

Data Updates and from which will be generated a higher-level Data Update. This capability to arbitrarily interconnect DtPs is the basis for the construction of a networked/hierarchical flow of Data Updates of our model.

As a service executing on top of PubSubAPI, the SDDS thus enables a distributed, mobile network of DtPs and DtCs for structured data/information sharing between application instances executing on either mobile or static nodes. And since SDDL handles all the details and problems concerning node mobility and intermittent connectivity, the information-sharing networks enabled by SDDS become completely independent of the underlying communication infrastructure. Each Data Provider can be deployed and activated/deactivated independently, depending on whether there is some DtC or other DtP interested in the corresponding information type. SDDS also supports the discovery and dynamic download of new Data Providers from a remote repository of DtPs.

SDDS has been implemented in Java as a part of the ClientLib. It has a single implementation that runs on the Android platform as well, making it suitable for mobile nodes. To use its services, the application must instantiate a new *StructuredDataDisseminationService* object by providing an instance of the *PublishSubscribeManager*.

There is an abstract class *DataProvider*, which must be extended to implement a new DtPs. These classes must be compiled in the platform in which they are suppose to run, even if they don't have platform-specific code. In most cases, where no platform-specific sensor of library is used, the same code can be compiled to both Java and Android.

The *DataUpdate* object must be created and populated by the DtPs to issue Data Updates. After generate a new DtU, the DtP just need to call the provided *publishDataUpdate()* method so the SDDS will deliver it to all interested DtC.

A *DataProviderRepository* is provided and works also as a Java ClassLoader. It dynamically loads and activates required DtPs. Since there are some differences between the Java and Android ways of dynamically load classes, the repository must be created by the application, providing the specific class loader of the corresponding platform. The repository automatically loads all DtP locally available and registers them at the SDDS, which will be listening to subscriptions from any node interested in those types of DtUs. Once a subscription is received, the correspondent DtP is activated to start producing DtUs

.

sdds://any/location?frequency=5s

Figure 15 – Information URI example.

Within SDDS a information is identified by an URI, for example ***sdds://this/location***, which represents the location of the local device. Figure 15 shows an example of an URI that are used by the subscribers. The blue part is the address of the device from which the information is needed; this may be *this* or *any* too. The green part is the desired information itself, represented by a keyword. In the query string (red part) it is possible to configure the Data Providers parameters, for example, setting its update frequency to five per second. The subscriber uses a URI to identify what information it needs.

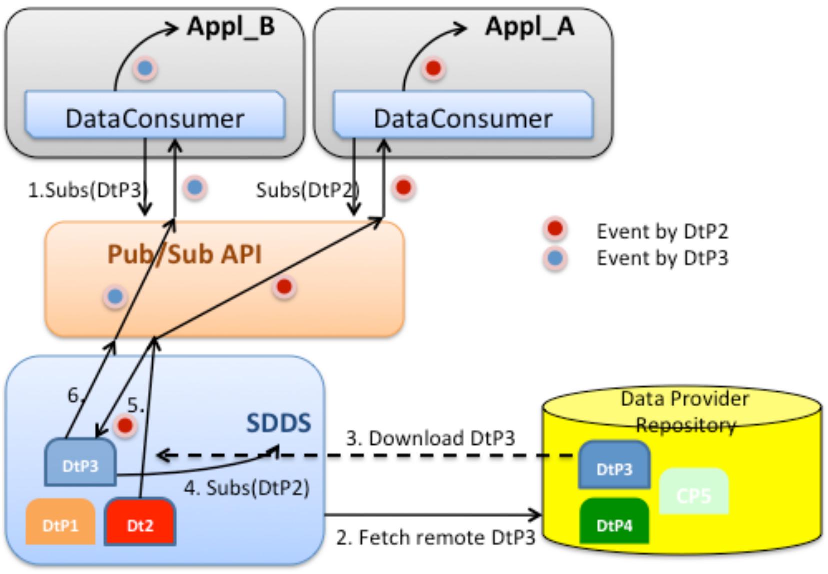


Figure 16 – SDDS interactions upon new context subscription.

Figure 16 illustrates the basic interactions between an application, the SDDS and the PubSubAPI: A *DataConsumer* of an application issues a subscription to PubSubAPI for some specific type of information, e.g., Battery level, through SDDS (step 1). Then, the SDDS searches for any locally deployed Data Provider that can produce the requested information. If none can be found locally, SDDS searches, downloads the corresponding DtP from the remote repository or others mobile nodes and activates it (steps 2 and 3). Once activated, the DtP delivers the created data to SDDS as a *DataUpdateObject*. SDDS adds some data to this object (e.g., the deviceID and timestamp), and publishes it through PubSubAPI (steps 5 and 6). This object is then delivered to all *DataConsumers* with matching subscriptions, which in turn pass it to the applications for specific handling.

6 Evaluation and Tests

Our model and middleware was designed and implemented in accordance with the desired features listed in Chapter 3. We studied the most promising researches and improved their ideas with some of our own. In this chapter we would like to evaluate our solution, presenting proof of concept applications, performance and stress test results and compare it with the related work.

6.1. Fleet Tracking and Management Application

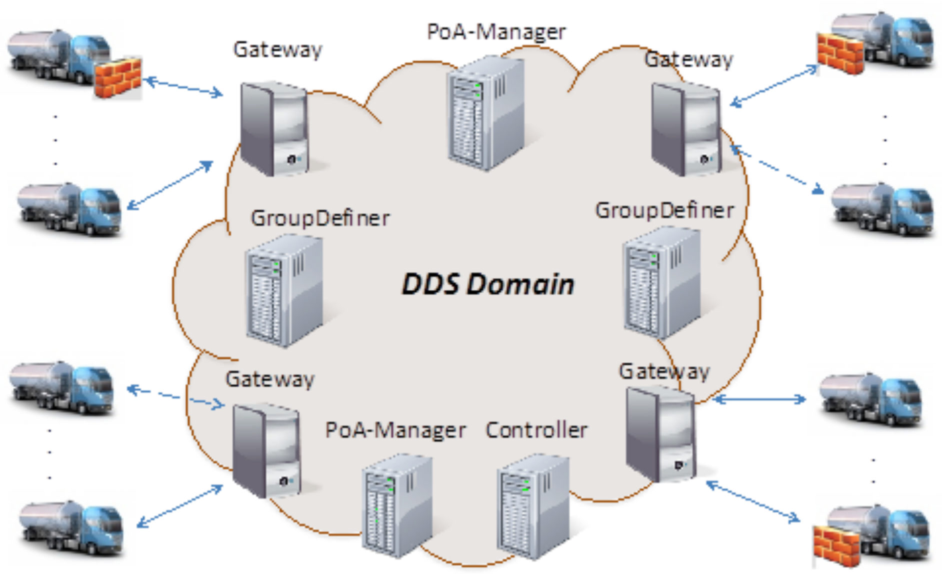


Figure 17 – Fleet Management Application architecture.

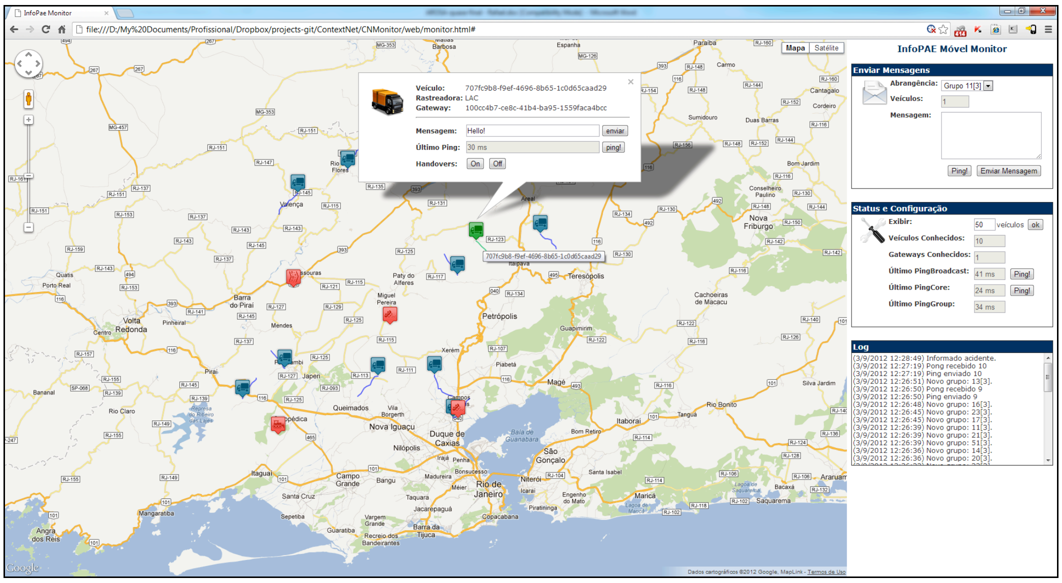
The ContextNet has been deployed in a real-world Fleet Tracking and Management application (InfoPAE Móvel) of a major gas distribution company, which operates throughout the entire country in Brazil. Using this application, the company's Operations Center is able to track trajectories of its trucks in real-time, in order to optimize the trucks' itineraries, to detect and notify obstructions or jams on roads, and to monitor the vehicle driver's actions (e.g., elapsed time on both planned and involuntary stops). Moreover, it does simple text messaging with drivers, to send them instructions or alerts, both individually as well as to subgroups of the vehicle's drivers, according to the country region they are currently located. For the communication with the vehicles, the company uses any of the four Brazilian cellular network operators, since one or the other operator(s) better serves each region of the country. Moreover, in each region, there are significant differences of connectivity quality (e.g., 2G vs. 3G) and extension of the wireless coverage. Thus, during a long journey, vehicles may experience several IP address changes, temporary data link disconnections (due to weak coverage, and caused by handover latency). Finally, in most cases their 2G/3G connections will be behind firewalls of the cell operators.

Figure 17 shows our application architecture, with all nodes in the SDDL core network (DDS Domain) and our Fleet Tracking and Management application. In the next sections we will present the implementation and performance test results as a proof-of-concept application for our middleware.

6.1.1. Implementation

Using the ContexNet as the middleware to implement this application, the mobile nodes are represented by the company's trucks. Once connected, the Data Provider at the vehicle sends up to 20 location updates (probed from the GPS sensor) every 30 seconds to the Gateway. This on-line tracking of all mobile nodes can enrich the quality of collaboration among the operator at the Fleet Management Operations Center (FMOC) and the drivers, and among the drivers themselves. Since all participants can be made aware of each other's location (in fact, it could be also other context information about the truck or its environment), it is possible to react immediately to any abnormal situation, and perhaps initiate a communication session with the drivers. For example, one could ask a driver why he/she has stopped or is traveling at low speed, and by such get informed about a traffic jam or an accident, allowing other drivers to choose a different route.

As part of the fleet management system we implemented another specific element, the Controller. The Controller runs at the FMOC and is used to display, in real-time, the vehicle's position on a map, as well as to send unicast, broadcast and groupcast messages to groups of vehicles. In the current version, the Controller is a Java Applet that interacts with a JavaScript to display vehicle positions, groups and text boxes for messaging in a Web browser window.



Veículo: 707fc9b8-f9ef-4696-8b65-1c0d65caad29
Rastreadora: LAC
Gateway: 100cc4b7-ce8c-41b4-ba95-1559faca4bcc

Mensagem: Hello! enviar
Último Ping: 30 ms ping!
Handovers: On Off

707fc9b8-f9ef-4696-8b65-1c0d65caad29

InfoPAE Móvel Monitor

Enviar Mensagens
Abrangência: Grupo 11[3]
Veículos: 1
Mensagem:
Ping! Enviar Mensagem

Status e Configuração
Exibir: 50 veículos ok
Veículos Conhecidos: 10
Gateways Conhecidos: 1
Último PingBroadcast: 41 ms Ping!
Último PingCore: 24 ms Ping!
Último PingGroup: 34 ms

Log

```
(3/9/2012 12:28:49) Informado acidente.
(3/9/2012 12:27:19) Pong recebido 10
(3/9/2012 12:27:19) Ping enviado 10
(3/9/2012 12:26:51) Novo grupo: 13[3].
(3/9/2012 12:26:50) Pong recebido 9
(3/9/2012 12:26:50) Ping enviado 9
(3/9/2012 12:26:48) Novo grupo: 16[3].
(3/9/2012 12:26:45) Novo grupo: 23[3].
(3/9/2012 12:26:45) Novo grupo: 17[3].
(3/9/2012 12:26:39) Novo grupo: 11[3].
(3/9/2012 12:26:39) Novo grupo: 21[3].
(3/9/2012 12:26:39) Novo grupo: 51[3].
(3/9/2012 12:26:36) Novo grupo: 14[3].
(3/9/2012 12:26:36) Novo grupo: 20[3].
```

Figure 18 – Fleet Management Application Controller.

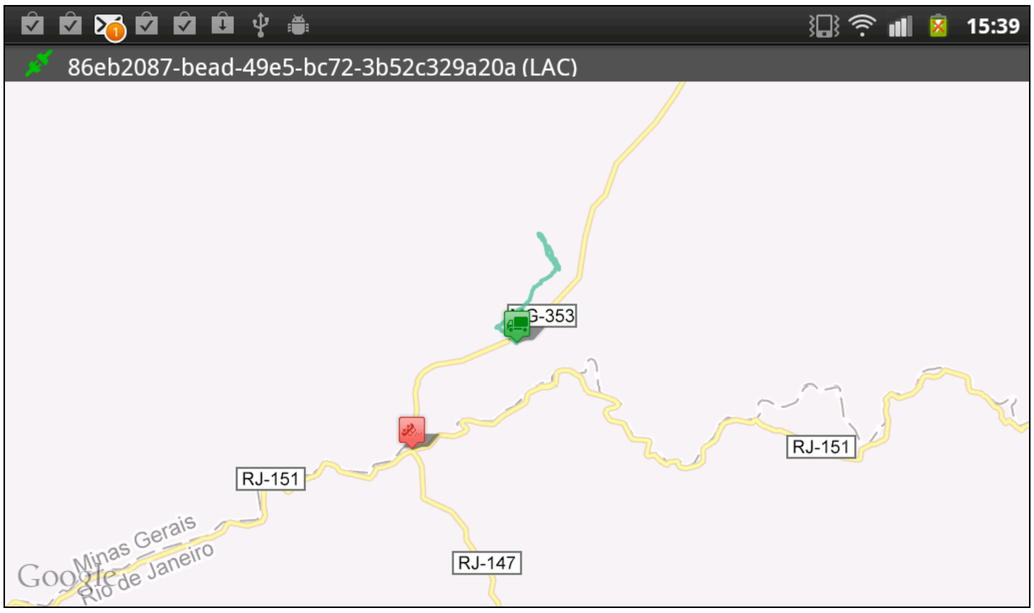


Figure 19 – Mobile Client App prototype running in Android.

Figure 18 shows a screenshot of the FMOC Controller (*InfoPAE Móvel Monitor*) browser window, with some vehicles (blue icons) with their traces, and some informed road problems/alerts (red icons) displayed on the map, as well as a “bubble window” for messaging with one specific vehicle (the green icon). On the right hand side, from top to bottom are: a section for editing and sending a message to a group of vehicles (with a group selector), a control panel for measuring round trip delays to individual vehicles or groups of vehicles, and a window displaying a log of message exchanges, respectively.

Regarding the mobile client for this application, so far we have implemented a prototype using the Android framework (version 2.3). This prototype uses the ClientLib in an Android's *AsynchTask* to connect to a Gateway (the first in the PoA-list), and is capable of sending and receiving simple text messages to/from any node, but most often, to the Controller. Using Android's *MapView*, it displays on a map the current vehicle's position (the green icon), as well as any red icon about a road problem/alert in its vicinity (see Figure 19).

6.1.2. Performance Tests and Results

We have tested our middleware in lab experiments and not in the real-world Fleet Management application, due to two main reasons. For one, our Controller and mobile client are still prototypes and do not implement all the required Vehicle Management functionality, and secondly, because we wanted to test our system with thousands of nodes, and doing such a large-scale deployment of the client software is currently not feasible. Therefore, we used a program to launch and simulate an arbitrary number of concurrent mobile nodes that connect to some Gateway and periodically send their position.

The main goal of the tests was to evaluate the ContextNet performance, in terms of communication latencies within the core network and on the Gateway-mobile links, both of them for unicast, broadcast and groupcast messages from the Controller to the mobile nodes, all in a real-world-like scenario.

We did two separated tests, one with all participants' nodes and simulated MNs executing in a local network (LAN) and another one with the simulated MNs connected through a remote link on the WAN. The local area test was primarily aimed to evaluate the ContextNet performance at serving a large number of mobile nodes and significant amount o data being exchanged, while the wide-area tests showed the reliability of the mobile communications using MR-UDP protocol, even in the presence of handovers. The PoA-Manager and handovers were active only on the WAN tests.

Even though we tested ContextNet with simulated mobile nodes, their communication behavior is very similar to the expected one of real mobile clients, except for the lack of mobile-initiated messaging, and moreover they use the same ClientLib/MR-UDP implementations. For example, in the WAN tests we also let the simulated MNs randomly disconnect from their current Gateway and try to connect to another Gateway. Therefore, the simulated MNs did in fact produce quite realistic traffic data, allowing us to measure the system's performance in high workload scenario, i.e., with huge volume of data exchanged between the mobile nodes and the core nodes. So, we believe that analyzing the system's performance graphs gives a realistic picture of the middleware's scalability and robustness.

6.1.2.1. LAN Tests

The main goal of the Local Area Network experiments was to evaluate ContextNet's performance under high traffic load of LocationUpdates (i.e., Data Updates), generated by thousands of mobile nodes.

Configuration and Simulation Parameters

Our mobile node simulation program, MN-Simulator, uses a thread pool of size 30 to indefinitely execute an arbitrary number of MNs, where each MN is scheduled to periodically send 20 simulated coordinates (pairs latitude, longitude) packed into the DataUpdate object to one of the Gateways. Thus, the total size of this LocationUpdate (LU) message is approximately 1 KB[7]. In addition to sending LUs, each MN also receives sporadic *ping* messages from the Controller, and immediately replies with a *pong* message.

The performance tests were executed with following system configurations and simulation parameters: (a) 2,000, 4,000, 6,000, 8,000 and 10,000 MNs connected to each Gateway; (b) one or two Gateways; (c) LocationUpdate frequencies of 2 LU/min, 4 LU/min and 10 LU/min; (d) one GroupDefiner.

Experimental Setup

To test the communication performance, in each test round we connected all simulated MNs to the Gateways and then sent unicasts messages to some MNs, broadcast messages to the Gateways on the DDS domain (Core) and broadcast messages to all MNs. For each type of message we calculated the round trip delay as the difference between the moments the message was sent and the moment the confirmation response was received.

Our hardware test setup was composed by 4 computers (virtual and real), 2 of them running Gateways and 2 others running the MN-Simulator. The GroupDefiner was running on one of the simulations machine. All machines were connected through a 10/100 Mbps switch.

We have run experiments with most of the simulation parameters explained in the previous section. However, due to memory and processing limitations of

7 Because the MR-UDP configuration for the tests carries 256 Bytes on each UDP packet, each LU is split to at least three UDP packets.

the machines executing the MN-Simulator, we were able to simulate at most a total number of 12,000 MNs performing 10 Location updates per minute.

Testing unicast and broadcast

The results are presented on Figure 20 and Figure 21. All round trip times are shown in milliseconds. For the sake of better legibility, subtitles were abbreviated, e.g., LU means Location Updates; 8,000v 2GW means a total of 8,000 MNs connected at 2 Gateways (4,000 at each GW). In all experiments we started to measure the delays only after all MNs were sending their LUs, and all results are the mean value of 5 measurements.

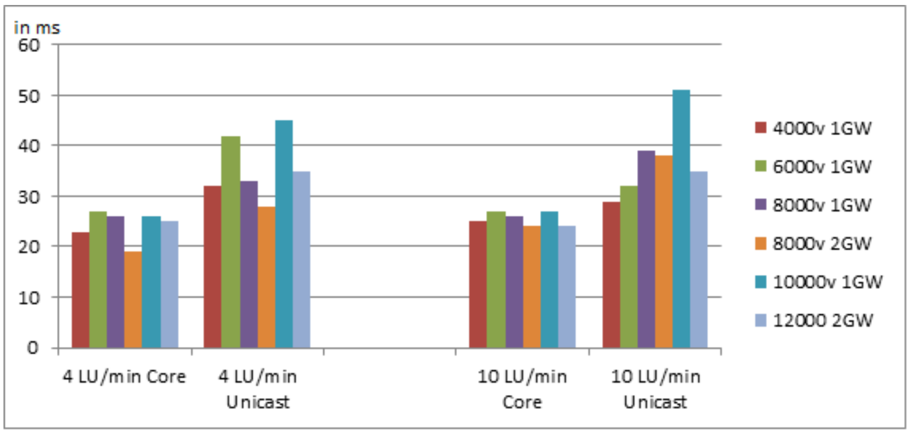


Figure 20 – Core and Unicast round trip delays.

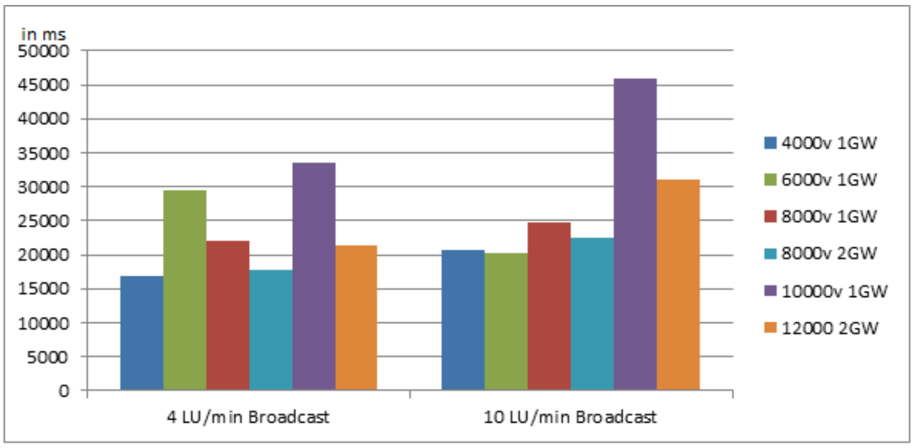


Figure 21 – Broadcast round trip delays.

As Figure 20 shows, the unicast and core round trip delays are very stable for all test parameters (20-45 ms). This suggests that our system is not yet overloaded. Unicast messages to any MN are delivered quite fast (up to 50 ms), and the ContextNet core network is yet far from saturation (< 20 ms), which means that it could handle a lot more messages. As shown by Figure 21, the broadcast delays are much higher (up to 45 sec), which is expected as all MNs must be contacted individually and their response must be obtained until the total round trip is completed. As mentioned before, we couldn't send a broadcast message to more than 10,000 MNs connected to a Gateway, because this caused a drop of connections during broadcast tests. We think that this is partially caused by the overload of the network interface, as all messages are sent almost instantaneously, probably causing a UDP buffer overflow. Additionally such storm of message causes a huge increase in CPU and memory resources on both the GW and MN-Simulator, causing the application to not working properly.

6.1.2.2. WAN Tests

In order to evaluate the performance of the middleware in an WAN environment with high latency connections and subject to intermittent connectivity and/or the occurrence of IP address changes (such as those experienced by mobile nodes connected by mobile network providers), we did the following experiments: we ran several performance tests involving 3 Gateways and 1 PoA-Manager executing in our lab, and several thousand simulated mobile nodes/vehicles launched in parallel on 4 to 5 remote machines served by different broadband ISP internet connections. We measured the Round Trip Delay of both unicast and groupcast messages to the MNs. Some experiments also included frequent handovers, both initiated by the mobiles and/or by the PoA-Manager, the latter aiming load balancing among the Gateways. For all these tests, the LU frequency was every 30 seconds (2 LU/min).

Experimental Setup

The experimental set-up was as follows: In our lab, the three Gateways executed on separate machines: a Dell PowerEdge server (3.0 GHz, 2x Dual Core), a PowerMac G5 (2.5GHz Quad-core with 8GB RAM), and a PC (CPU Core i5 with 8GB RAM), while the PoA-Manager and a GroupDefiner executed on a separate PC. All these machines were connected to a 10/100/1000-Mbps switch. This switch, in turn, was connected to a 10/100-Mbps switch at the router serving the Internet connection of our lab. At the remote side, the machines were diverse, but all were connected via wired Ethernet to the ISP modem or router. Before executing the experiment, all home testers measured their effective uplink capacity, (which was always the range 0.25 to 0.9 Mbps) and downlink capacity (in the range 1.01 Mbps to 9.56 Mbps). We chose not to use WiFi wireless network, as this would create a less realistic simulation scenario, since all simulated MNs would be competing with each other for a single wireless IEEE 802.11 connection, which is not collision free, as opposed to what happens with real-world Edge or 3G connections. However, we emulated the intermittent connectivity of real-world wireless connections by making the simulated MNs randomly their MR-UDP connections and reconnect to a new Gateway. Also, there was very little interference of the Internet connection usage by other applications on the remote machines.

Testing unicast and broadcast without handovers

In these experiments, the MN-Simulator program (let's denote by *MS-i* the MN_simulator program launched at remote/home machine i) initially connected all the simulated MNs to a single Gateway, but right after each of them established the MR-UDP connection with this Gateway, it received (only once) a PoA-List of size 3, containing the IP addresses and ports of all the three gateways running in our lab (which was used for the handover tests – later in this section). In this experiment, we turned off the load-balancing function of the PoA-Manager, since we wanted to evaluate exclusively the ContextNet's performance with mobile-initiated, spontaneous handover, i.e., without any interference/overload caused by mandatory handover requests by the PoA-Manager.

Table 2 shows the round trip delays (RTD) of the unicast messages for three total amounts of simulated MNs executing at the remote machines. Since the Internet connectivity and the remote machine's capacities were so different, we measured the mean RTD time for each vehicle simulation programs separately.

Total nr of MNs	MS-1	MS-2	MS-3	MS-4	MS-5	Global mean
1000	108	67.80	70	67.2		78.25
4123	115.8	86.20	84.4	87		93.35
7174	98.8	68.60	77.4	67.4		78.05

Table 2 – Round Trip Delays of Unicast to MNs of each home machine (in ms).

The lower value for unicast RTD for 7174 simulated MNS was probably caused by a sudden performance boost in the throughput of the ISP up/downlinks at one or more of the remote Internet connections. It also indicates that the increased number of clients does not yet affect the communication performance. The total number of MNs is not a multiple of 4 because during the parallel launch and connection of >1000 MNS to a Gateway some of the MR-UDP connections failed to be established, and our vehicle simulation program was not conceived to retry all failed connections several times.

In this same test run, we also measured the RTD of a broadcast to – and reply from – all 1000 MNs, executing on the 4 home machines, which was only 47,1 seconds. Since the broadcast incurs in too much instantaneous communication load at the MN-Simulator program and their Internet connections, we were only able to execute it for 250 MNs per machine.

Tests with mobile-initiated handovers

In order to test the performance of SDDL with spontaneous, mobile-initiated handovers and with intermittent connectivity of the mobile nodes, we added – just for this experiment – a new message to the system, the Handover Test messages (HT), and modified the PoA-Manager and the MN-Simulator program, accordingly to do also the following:

Every 3 minutes, each MNs decides if it will disconnect from the current Gateway and reconnect to another Gateway, chosen randomly from its PoA-list. This decision is controlled by a handover probability (HO_P), which we varied from 0% to 15%. Whenever a MN starts a handover, it first closes the current MR-UDP connection, and then requests a new MR-UDP connection to the newly chosen Gateway, i.e., for some short period of time – a few ms – the simulated MN is entirely disconnected from any Gateway. Each handover is printed at the terminal console. Each MN also accepts the HT message and increments a global counter, which is also printed at the console.

The purpose of the HT message is to test the reliable delivery of messages to the MNs during a handover/disconnection. It is sent by the PoA-Manager immediately after it receives a "connection closed" message from the corresponding Gateway. Since the mobile node is disconnected, the non-delivered messages are received by the MTD service, and later forwarded to the new Gateway where the MN reconnected. Thus, we wanted to check, at each MN-Simulator program, if the total number of received HT messages equals the total number of performed handovers by the MNs, i.e., if the MTD service had replayed all the non-delivered unicast messages, or if some unicast message had been loss during the handover.

Table 3 shows the mean values of round trip delays (RTD) of unicast messages for four combinations of total number of MNs and handover probability (HO_P), again, presented separately for each home machine.

Total nr of MNs/HO_P	MS-1	MS-2	MS-3	MS-4	MS-5	Global mean
1800/15%	103.6	72	65	61.2	70.2	74.4
3979/5%	93.2	68.2	84	63	73.4	76.36
5812/5%	112.6	79.2	102	70	92.2	91.2
7815/10%	79	58.8	59.6	50.4	334.8	116.52

Table 3 – Round trip delays of unicast messages (to each home machine) under different handover probabilities (in ms).

From this data, we can see two things: (i) a higher handover probability, does not necessarily increase the overall RTD of unicast messages, showing that the retransmissions by the MTD and the disconnection management by the Gateways only affect the message delivery times of the migrating mobile nodes; (ii) for a same handover probability, e.g., 5%, larger number of total mobile nodes does slightly impact on the increase of the overall message RTD.

When comparing the data of Table 2 and Table 3(for approximately 4000 MNs), it is interesting to notice that the unicast RTD are similar, and even decreased a little bit in the experiments with low-probability mobile-initiated handovers.

Total # MNs/HO_P	MS-1	MS-2	MS-3	MS-4	MS-5
1800/15%	2.4	1.7	4.9	3.1	1.5
3979/5%	4.9	5.9	2.5	3.0	6.2

Table 4 – Percentage of “missing” HT messages after stopping the MN-Simulator programs.

Concerning the delivery of HT messages, there is a natural delay due to the fact that the MTD service only resends non-delivered messages to the mobile nodes, after the connection establishment is announced by the new Gateway. And since we did not implement the MN-Simulator to stop doing handovers after some time, at the end of the simulation, there is always a gap between the last announced handover, and the corresponding delivery of the HT message, and this gap obviously increases with the number of MNs, and their probability of doing handovers. Table 4 shows the percentage of “missing” HT messages at the end of the simulation for the tests with 1800 and 3979 MNs. However, when examining the output logs of the MN-Simulator, almost all the HT messages (of past handovers) were delivered. This raises our confidence that ContextNet supports reliable delivery of messages in the presence of handovers between Gateways.

Tests with groupcast messages

The purpose of this test was to measure the RTD of groupcast messages (including the corresponding acknowledgements by all group members), for different sizes of groups, where the group members were simulated by MN-Simulator programs (MS-i) executing on the remote machines served by the different ISPs. Since we did this experiment on a different day and from other remote machines, we named them MS-6 to MS-11 to make clear that the RTD times of this and previous experiments cannot be compared. In this experiment, common ping delay was around 25 ms (except for MS-11, that was 444ms), and down- and up-links varied between 1.59 – 1.2 Mbps and 0.93 – 0.33 Mbps, respectively. It should be noted that MS-11 was a machine connected in Europe, and therefore its RTD is so much higher than the other vehicles executing on Brazilian machines. For this experiment we turned off the induced mobile-initiated handover behavior of the simulated MNS (HO_P=0), i.e., they would only switch to another Gateway if their MR-UDP connection in fact failed.

The group size is approximate, as it was determined by the GroupDefiner using a mod operation (e.g., x%100) over the least significant byte of the MN-identifier, which is a randomly generated UUID. Thus, in the Gr-10%, the group had approximately 10% of 5795 MNs, etc. Recall that in all test runs, the core nodes were also busy processing the LU messages sent every 30 seconds by each MNs.

Vehicles/Mode	Group size	MS-6	MS-7	MS-8	MS-9	MS-10	MS-11	Gr-cast RTD
5794/Unicast	0	100	59,6	58,4	59,2	50	289,4	
5795/Gr-10%	579	🟩	🟩	🟩	🟩	🟩	🟩	19720,60
5430/Gr-25%	1358	🟩	🟩	🟩	🟩	🟩	🟥	66437,80

Table 5 – Round trip delays of unicast and groupcast messages (in ms).

Table 5 shows the mean RTD times of 5 measurements, both for the unicast and the groupcast communication modes. The color of the field indicates which remote machines actually executed vehicles that participated in the group-cast (red means: not used). The numbers reveal that the mean RTD time for the estimated 579 and 1358 group members is only 19.7 and 66.4 seconds. This suggests that a one-way groupcast message is probably delivered to the entire group members is 40-70%-fraction of this time. Moreover, although we don't know how many group members were actually executed by MS-11, its longer ping delay certainly contributed to the total increase of the RTD in the Gr-10% experiment. As mentioned in the previous sections, we also tested and measured the RTD of a broadcast to 1000 MNs, and the obtained results for 1000 and 1358 deliveries and replies seem to be consistent.

6.2.
Data Dissemination Tests

The data dissemination tests described in this section aim to isolate and stress specific situations to evaluate the middleware's performance so to certify specific features such scalability, reliability, on-time dissemination, intermittent connection resilience, etc. Different from the tests with the Fleet Management Application, the data dissemination tests uses semi-automatic and hypothetic applications implemented with the SDDS API designed specifically for the tests. Hence they do not work as a proof of concept application, but aim to stress the middleware in situations that are unlikely to happen in real world and so find some of the current implementation limits.

Experimental Setup

All the test experiments were performed in our lab using five notebooks connected to a 10/100/1000 switch. One of the machines is a MacBook with a 2.3GHz Intel Core i5 processor and 8GB of RAM. The other 4 machines are PCs equipped with 2.6GHz Intel Core i5 processor and 4GB of RAM running Ubuntu 12.04 LTS. However, in the each PC we used a virtual machine with 2GB of RAM running the same Ubuntu 12.04 LTS. It is worth to mention that the use of virtual machines may reduce the overall performance, especially by using such a small amount of memory. However, we chose to use virtual machines since we decided to use them for the sake of uniformity and the ability to deploy new machines as needed, a means to test scalability of the middleware. For the experiments with mobile devices, we used a Motorola Xoom 2 Media Edition, running Android 4.0.4.

We chose to run the experiments with real simulations – i.e., experiments with the entire software stack, and where only the Data Update data (i.e., the position update) was emulated – instead of using network simulation software, such as ns-2 or OMNET++, because it would be we needed to test all implementation layers of our solution, as well as test with real mobile. Also, we think that such real-world-like experiments, testing specific parts of the system, can give a better picture of the system's behavior and reveal the real middleware implementation's limitations.

Experimental Application

To run the experiments we implemented a TesDataProvider that provides the *testDataUpdate* information. This DtP can receive three configuration parameters: the **total** number of DtUs that should be sent, the **freq**uency (in messages per second - Hz) in which the DtUs should be produced and the number of times (**repeat**s) that the test is expected to run. The DtP class file was placed in the local repository directory and is automatically loaded by the SDDS when the application starts. When a subscription is received by an instance of SDDS, the DtP is activated locally. The first message sent by the DtP is a *testStarted* DtU (which of course, would not exist in a real DtP) with all test configuration parameters. Through this message the corresponding remote DtC will become aware of what will be the test. Next, the DtP will start producing the **total** number of *testDataUpdate* DtUs in the frequency **freq** specified by the test, and will close the DtU data generation with an *endTest* DtU message.

The DtC was implemented through the TestConsumer class, which is used to subscribe to the *textDataUpdate* in SDDS. When the *testStarted* DtU is received, the DtC starts a new test round, resets its internal number of DtU-received-counter and saves the time the test started. Subsequently, when receiving each DtU, the DtC increments the corresponding counter and sets the time of the last message received. When the *endTest* is received, the DtC groups the information from the test (i.e., test round, number of expected DtUs, number of received DtUs and total time elapsed) in a structure that is added to a list that will maintain the results in memory until the application finishes, writing all test results to disk. This in-memory list is shared by all DtC that the test application may have. We chose to maintain the results in memory until the end of the test to avoid decreasing the application performance due to file output.

The aforementioned programs were the basic DtP and DtC implemented for all the tests. An application that receives command line parameters controls the components. Additional components implemented for specific tests will be explained when appropriate.

As for the experiments, explained in what follows, we ran each test at least five times and collected the results in each phase. Thus, the results in this section will present the average result and the standard deviation after analysis of the data of each test round.

6.2.1. Test 1 – Dissemination Rate for One Information

The goal of this test is to evaluate how the system performs when one DataProvider (DtP) is generating DataUpdates at high frequency for some interested DataConsumers (DtC). We want to evaluate the maximum sustainable data flow rate of DtU messages.

Part 1 – Finding one information production and dissemination rate

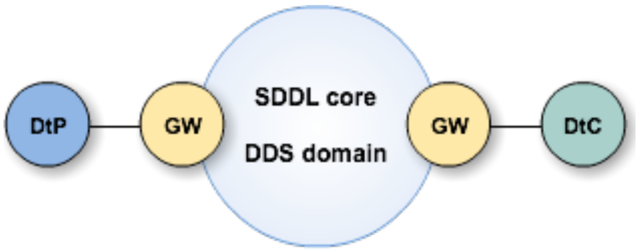


Figure 22 – Architecture of Test 1 – part 1.

In this first part we tested with only one DtP and one DtC, each connected to one specific GW, as shown Figure 22. First we connect the DtP and then the DtC. We ran tests with 100, 1,000 and 10,000 DtUs with the frequency set to 1000 Hz (one message each ms). This frequency is very high and is very unlikely that the system would perform in this rate. However, this forces the performance to its maximum, including testing the MR-UDP's buffers and reliability, which would slow down the DtU throughput to real maximum due to its simple control flow feature. Table 6 summarizes the test parameters.

TOTAL GW	TOTAL DtP	TOTAL DtC	TOTAL DtU	FREQUENCY
2	1	1	10/1,000/10,000	1,000 Hz

Table 6 – Parameters of Test 1 – part 1.

Table 7 shows the test results. The first column (*TEST ID*) identifies the test round (i.e., the test with 100, 1,000 or 10,000 DtU). The column *TOTAL DtU* displays the total number of DtU produced in the DtP for all tests rounds. The next column (*LOST DtU*) shows the total number of lost DtU that didn't arrive at the DtC. The *AVG DtU/s* column shows the average number of DtU per second received in the DtC and the column (*STDDEV DtU/s*) presents the standard deviation for the average number of DtU per second.

TEST ID	TOTAL DtU	LOST DtU	AVG DtU/s	STDDEV DtU/s
100 DtU	500	0	97.64	5.92
1000 DtU	5,000	0	110.48	1.78
10000 DtU	50,000	0	113.92	0.82

Table 7 – Test 1 – part 1 results.

As the results of all the test shows, the middleware can deliver around 110 Data Updates per second, which is a good performance result. Since ContextNet was designed to disseminate context information, the values are more than well suited for this kind of information dissemination. In real world applications, most of the types of context information will not generate updates at this rate. For example, one of the context types that could generate updates at high frequency is geographical location, but current applications only update this information each tens of few seconds. So, if we want the location to be updated at every second or twice a second, our middleware would still perform very well. Moreover, Table 7 also shows that the **communication is reliable**, since no Data Updates were lost, even when 50,000 of them were generated and delivered at the rate of 113,92 Hz. In fact, this characteristic was also noticed in all other tests explained in this section. Combining these numbers with the round trip delays of pings presented in the previous section, we can also deduce that ContextNet indeed is suitable for **on-time dissemination** of context information. The standard deviation is very low, which points to a representative average value.

In Table 7 it is also noticeable that the performance increases with more Data Updates being generated. This can be explained by analyzing the values together with the CPU and memory resources usage on the experiment machines. Unfortunately we didn’t have a good means of gathering the resource usage during the tests, so as to present them together with the results. However, the system monitor of the machines revealed that at the start of the test a burst of CPU usage and memory allocation occurs (specially for the following tests), causing the system to perform a little slower in this situation. After this initial burst, however, the system (i.e., all the software components like the GW and simulation applications) usually accommodates itself and begins working more smoothly. This also explains that the standard deviation is lower in the 10,000-context-updates test, which happened because the system had more time than in the other test configurations (and more data to process) to reach the state of resource accommodation.

In the next sections we will use the same tables structures to show the test configuration parameters and results. However, we will exclude the *TOTAL DtU* and *LOST DtU* columns from the results, because we think these columns do not present any benefits for the tests performance results. Their main purpose is to show reliability, which already was demonstrated.

Part 2 – Finding one information consumption rate

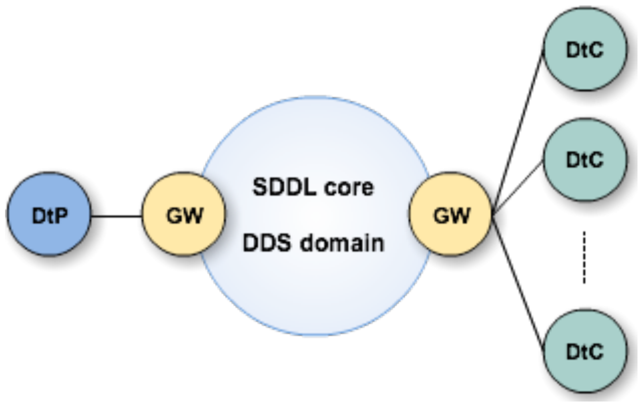


Figure 23 – Architecture of Test 1 – part 2.

In this part in chose an intermediary value of Data Updates and increased the number of Data Consumers gradually with the purpose to find for how many Data Consumers the system would still perform disseminating 110 DtU/s. Table 8 and Table 9 shows the test parameters and results, respectively.

TOTAL GW	TOTAL DtP	TOTAL DtC	TOTAL DtU	FREQUENCY
2	3000	10 ~ 100	3,000	1000 Hz

Table 8 – Parameters of Test 1 – part 2.

TEST ID	AVG DtU/s	STDDEV DtU/s
10 DtC	111.44	2.57
25 DtC	111.74	3.07
30 DtC	111.08	2.91
50 DtC	67.91	1.97
100 DtC	33.28	1.27

Table 9 – Test 1 – parte 2 results.

Analyzing the results, we found that 30 Data Consumers is a good limit to maintain the 100 DtU/s rate. So, we assume that one GW can deliver around 3,300 DtU/s for any number of DtP and DtC. We also ran the test with 1,000 DtC, but it was not possible to finish the test because the CPU usage on both GW and DtC machines went to its top, and therefore MR-UDP starts dropping connection.

A high-usage of machine resources was also the case in the 50 DtC and 100 DtC, but the system could still deliver a solid performance, which we seem fairly enough for the majority of current real world context-aware applications and sensor probing rates. However, it is desirable to operate the system in good conditions of machine resources usage, to avoid undesirable behavior. Therefore, from this point on, we assumed that each GW delivers 3,000 DtU/s. And to evaluate the scalability of the system we added another GW to the SDDL core and connected another set of 30 DtC to this new GW. Then we ran the same test to check if our system reaches the same DtU delivery rate for 60 DtC. Figure 24, Table 10 and Table 11 shows this test architecture, parameters and results.

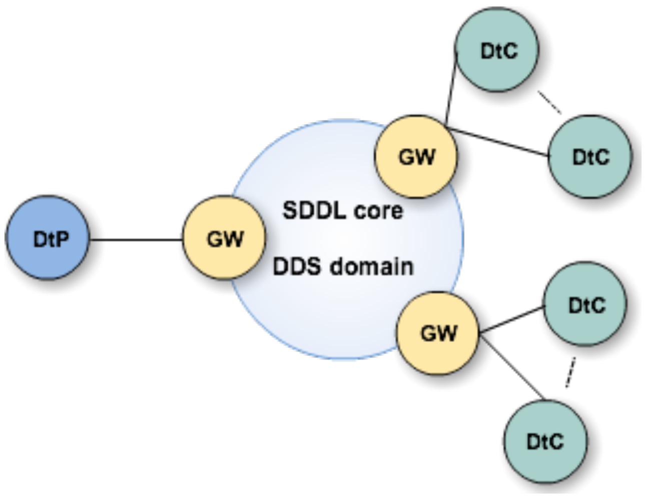


Figure 24 – Architecture of Test 1 – part 2, now with 3 Gateways.

TOTAL GW	TOTAL DtP	TOTAL DtC	TOTAL DtU	FREQUENCY
3	1	60	3,000	1 Hz

Table 10 – Parameters of Test 1 – part 2 with 3 Gateways.

TEST ID	AVG DtU/s	STDDEV DtU/s
60 DtC / 2 GW	111.65	6.56

Table 11 – Test 1 – parte 2 results with 3 Gateways.

As Table 11 shows, when we added another GW with more 30 DtC the system maintained the same rate of 110 DtU/s in each GW. This seems to be an indicator of our middleware's **scalability**. One could ask why we did not add more Gateways be sure about the system's scalability. The reason is that since DDS uses multicast or broadcast within SDDL core, the DDS messages received by some GWs are not copied to be delivered to additional GWs. Thus, adding more GWs is equivalent to just add more listeners for the DDS messages, and these listeners will not impact the overall system performance.

We think 3,000 DtU/s for each GW is already a pretty good number. And we chose it for the other tests because we wanted to ensure a “fair use” of CPU and memory resources in our notebooks. Thus, we believe that if the Gateways were deployed on more powerful server machines the overall performance would most probably increase substantially.

In both parts of Test 1 we measured a 45Mbps data transfer rate in the network adapter of the GW when it was delivering 110 DtU/s to 30 DtC.

6.2.2. Test 2 – Feasible Dissemination of Context Information

The goal of this test is to evaluate the limit of 3,000 DtU/s per GW to deliver an adequate number of context information at a rate that could happen in a real world application. We still deploy only one DtP, but now producing DtU at a lower rate but for much more DtC. The architecture of this test is the same as the one of Test1 – Part 2, but the parameters are different and are shown on Table 12.

We again have divided the test in three parts, but this time without big changes in the architectural organization of the components. In the first and second parts we used three GWs, one receiving DtUs from the DtP, and the other two GWs serving 2000 DtC each. The frequency of DtU is modest, with only 2 or 1 Hz, as indicated in the table. The results are shown in Table 13.

PART	TOTAL GW	TOTAL DtP	TOTAL DtC	TOTAL DtU	FREQ.
Part 1	3	1	4,000	30	2 Hz
Part 2	3	1	4,000	60	1 Hz
Part 3	2	1	3,000	5	1 Hz

Table 12 – Parameters of Test 2.

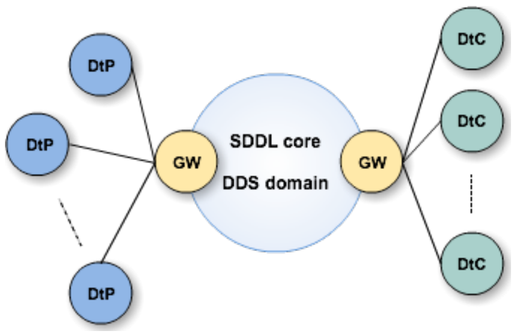
TEST ID	AVG DtU/s	STDDEV DtU/s
Part 1	0.50	0.01
Part 2	1.00	0.03
Part 3	0.61	0.03

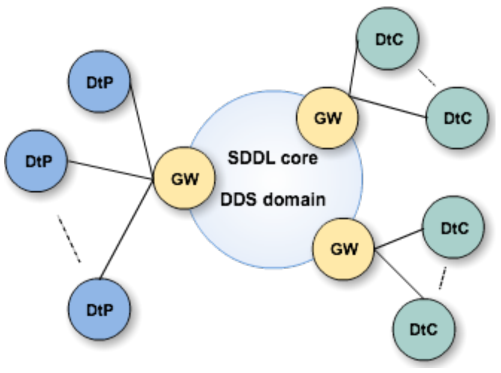
Table 13 – Test 2 results.

In parts 1 and 2 the average of DtU/s perceived in each of the 4,000 DtC was exactly the same as the DtU production frequency from the DtP. Thus, we can see that the system was preforming in perfect condition delivering the DtU on-time. In the Part 3, however, we had 3,000 DtC connected to a single GW, which overloaded the machine resources and degraded its performance. But it still delivers DtU at a good rate equally for all DtC, as indicated by the standard deviation.

6.2.3. Test 3 – Concurrent Dissemination of Information

In this test we evaluated the middleware performance with the presence of multiple DtPs disseminating the information to multiple DtCs. This situation, as shown in Figure 25 (b), is much alike to what will happen in real world deployment.





(a) (b)

Figure 25 – Test 3 architectures (a) with 2 Gateways and (b) with 3 Gateways.

PART	TOTAL GW	TOTAL DtP	TOTAL DtC	TOTAL DtU	FREQ.
Part 1	2	50	2000	10	1 Hz
Part 2	3	50	4,000	10	1 Hz
Part 3	2	50	1500	100	1000 Hz
Part 4	3	50	3000	70	1000 Hz

Table 14 – Parameters of Test 3.

We ran the tests with 50 DtP producing 50 different types of information. As for the DtC, we have divided the test in parts, each with different parameters, as shown in Figure 25. Each DtC subscribes to only one information type that was chosen randomly at the start of each test. Using the Java Random class it is guaranteed that the pseudo-random generation of numbers (i.e., numbers 1 to 50, each one representing one DtP) will distribute each number almost equally, generating the division of each DtP to a similar number of DtC. In our tests, it is not necessary to have the same number of DtC for each DtP. However, it is important to have at least one DtC for every DtP, because we want that the DtU of each DtP to be disseminated. Anyway, after the analysis of test results, we can confirm that every DtP was activated in all test parts. The results are displayed in Table 15.

TEST ID	AVG DtU/s	STDDEV DtU/s
Part 1	1.06	0.06
Part 2	1.09	0.05
Part 3	4.93	2.38
Part 4	79.93	14.10

Table 15 – Test 3 results.

Analyzing the results we can see that in Part 1 and 2, where the DtU generation was at 1 Hz, the middleware has disseminated the DtU in the same frequency of its production, meaning that the system was working in fair conditions. The first two tests emulate real world behavior with a production rate of 1 DtU per second. In the following tests (i.e., Parts 3 and 4), however, we have forced the middleware to disseminate the DtU in the most highest possible production rate (1000 Hz) for all 50 DtP. As we can see in Part 3 results, the test seems to be not so successful, as we were expecting a 110 DtU/s rate (the same from Test 1) and only got around 5 DtU/s. But, it is important to point that in Part 3 the information production of all 50 DtP initiated almost at the same time, which means that all 50 DtP were producing 110 DtU/s. Therefore, the actual production rate of DtU was around 550 DtU/s and that was been disseminated for 1500 DtC on a single GW. Thus, this means that the Gateway was disseminating around 7300 DtU/s to all its 1500 DtC (i.e., the number of DtC multiplied by the average reception rate). Moreover, let's suppose that all DtC are condensate in just one that counts for 50 DtC (because it receives 50 different information types). Now, if we multiply the number of DtC by the average DtU dissemination rate we get 250 DtU/s and if we subtract the standard deviation from the average and multiply it by the number of DtC we have 50 multiplied by 2.55, which results in 127.5 DtU/s, a number that is consistent with the DtU/s dissemination rate observed in the previous tests. Additionally, if we subtract the standard deviation from the average and multiply by the number of DtC we get 3800 DtU/s, a number equally consistent with previous test results.

To avoid that all DtP initiate the production of DtU at the same time we did a small modification in the test application for Part 4. Here, each DtP waits 1 second after the another DtP started to produce information and lower the production rate of DtU to 70 Hz. With this setup we wanted that each DtP takes 1 second to produce all its 70 DtUs exclusively using the network and then stops using the dissemination network for other DtP. So, each test round ran for around 50 seconds, with a second for each DtP. As results shows, since at any given time only one DtP was producing information for approximately 30 DtC (assuming that the 1500 DtC were equally divided to the 50 different DtP, which, actually, was not), the average dissemination rate was close to 80 DtU/s. Taking into account the standard deviation value (adding it to the average), we can correctly suppose that some of the DtU was disseminated in 100 DtU/s, as we could notice in the raw test results (i.e., the text output file generated by the simulation application). However, it is very important to point that we did not have control of how many DtC was subscribed to each DtP. Also, we could not guarantee that the waiting time of 1 second between DtU bursts was precisely adopted by the application. Moreover, even if the waiting time was respected, it did not mean that each DtP could disseminate all its information in 1 second, especially if one of this DtP had a number much greater than 30 DtC, and if, for example, a DtP had 40 DtC it will last more than 1 second to disseminate all its information and will use a little of the time that we reserved for another DtP, which will decrease the perceived performance. Another difference of part 4 is that it has two Gateways with 1,500 DtC each, which is twice the number of DtC in Part 3, but still presenting consistent values with all previous test, which again indicates our model and middleware scalability.

Thus, comparing the results of tests results of Part 3 and Part 4, we can observe that the system shows a uniform and scalable behavior, always providing a consistent value of around 3,000 DtU/s for each Gateway. So, if each DtP could use the network exclusively for a small amount of time to disseminate its information it could disseminate at 110DtU/s and 6,600 DtU per minute. If we have a real world DtP that produces a new DtU every 2 seconds and if we could orchestrate the production and dissemination on each DtU, in one minute we would have room for 220 DtP (during 2 seconds we would have 220 DtU and each DtP would produce 1 DtU) to produce 30 DtU each (in one minute) that could be disseminated to 6,600 different DtC (30 DtC to each DtP and because the GW can disseminate 6,600 DtU at every 2 seconds), in a total of 198,000 DtU in one minute. This with only one gateway with 220 DtP that produces a DtU every 2 seconds to 6,600 different DtC. In a real world this situation is very unlike to happen.

6.2.4. Test 4 – One Hierarchical Information Dissemination

So far, we have tested only a Data Sharing Network with primary Data Providers. For this and the following tests we have implemented a new, secondary DtP provider called *HierarchicalTestDataProvider* that subscribes to the primary *TestDataProvider* and forwards any DtU as soon as it receives it to the DtC, which will subscribe to the *HierarchicalTestDataProvider*. Figure 26 illustrates the architecture of this test. As can be seen, for the sake of clarity we have removed the core elements (GWs) from the images and show only the SDDS elements (DtPs and DtCs). The test configuration parameters can be found in Table 16. In this test we deployed one GW serving the primary DtP, one GW for the secondary (*HierarchicalTestDataProvider)* and the third gateway to the DtCs.

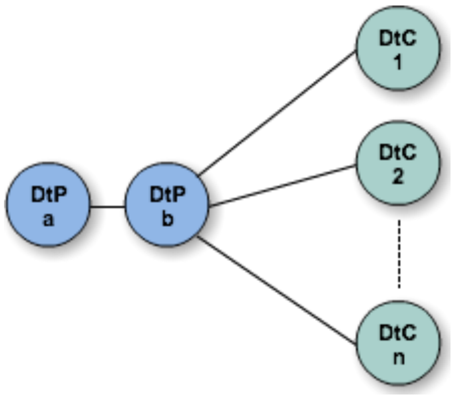


Figure 26 – Test 4 architecture.

TOTAL GW	TOTAL DtP	TOTAL DtC	TOTAL DtU	FREQUENCY
3	2	30	10,000	1 ms

Table 16 – Parameters of Test 4.

The goal of this test was the same of Test 1, i.e. to evaluated the highest possible dissemination rate, but now with a three-level hierarchical network. And as can be seen from the results in Table 17, this information flow indirection incurred in no overhead, and the middleware was capable of data dissemination of around 110 DtU/s.

AVG DtU/s	STDDEV DtU/s
109.32	4.55

Table 17 – Test 4 results.

6.2.5. Test 5 – Multiple Hierarchical Information Dissemination

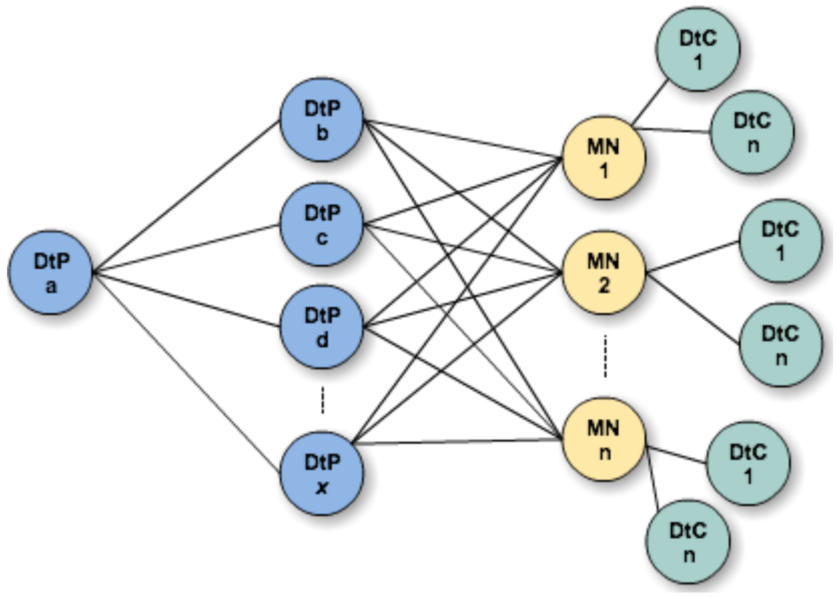


Figure 27 – Test 5 architecture (a).

In this test we want to combine the Tests 3 and 4 to evaluate the dissemination of multiples hierarchical DtP to multiples DtC, as shown figures Figure 27 and Figure 28. In the figures of the architectures we now show the mobile node because the test is a little different than the others. In the previous tests, each DtC was located in one single MN, so there was one DtC per MN (which means one DtC per connection to the GW). This time, however, each MN will have once DtC for each subscription. The test parameters are shown in Table 18. In parts 1 and 2 there was only one primary DtP and 100 intermediaries (hierarchical); in parts 3 and 4 we had 50 primary DtP and each one of the 100 intermediaries DtP was subscribed to 2 different primaries, so each intermediary DtP produces DtU twice as fast then the primaries, because each DtU received is replicated upon reception. The core components are used as the previous section, as will be for the next tests: one GW for the primaries DtP, another for the intermediaries and the third for the DtC.

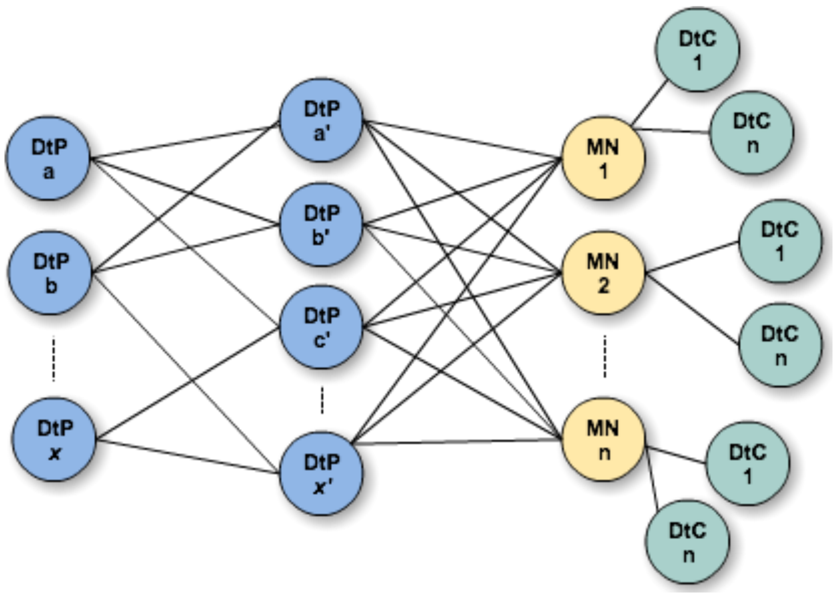


Figure 28 – Test 5 architecture (b).

PART	TOTAL GW	TOTAL DtP	TOTAL DtC	TOTAL DtU	FREQ.
Part 1	3	101	5,000	10	5 s
Part 2	3	101	5,000	10	4 s
Part 3	2	150	5,000	40	10 ms
Part 4	3	150	5,000	40	1 ms

Table 18 – Parameters of Test 5.

Please note that there is 5,000 DtC because each one of the MN has one DtC to each of the 100 intermediaries DtP. The test results are presented in Table 19. As we can see, the parts 1 and 2 ran smoothly, with the DtU being delivered in the same frequency as was produced. In parts 3 and 4, however, the standard deviation is not so low. This happens because for these test we ran 7 rounds to get the average value. This is why we set each DtP to wait 1.2 seconds to start after the initialization of the other. This causes the first test and the last to happen without all DtP initialized. In the raw results we had a more stable value, but still very close to the global average presented in the results. If this test we want to show that the middleware is capable to maintain its performance even with a arbitrary number of DtP and DtC creating a \ complex transformation tree and will still present consistent performance values.

TEST ID	AVG DtU/s	STDDEV DtU/s
Part 1	0.20	0.00
Part 2	0.30	0.01
Part 3	29.00	9.55
Part 4	23.24	13.22

Table 19 – Test 5 results.

6.2.6. Test 6 – Dissemination to Mobile Devices with Intermittent Connectivity

For this test we will repeat the Test 1, but now with the DtC deployed on our Android mobile device connected in our lab Wi-Fi network. In specific parts we also had a DtP installed on an Android, however running in the Android emulator. The DtP and DtC Java implementation, and the SDDS, are the same for the Android. Though, we needed to implement a very simple Android application just to connect to the GW and start the SDDS. The test parts parameters are specified in Table 20 and the results in Table 21.

PART	TOTAL GW	TOTAL DtP	TOTAL DtC	TOTAL DtU	FREQ.	TOTAL DISCONNECTIONS
Part 1	2	1	1	2,000	1 Hz	0
Part 2	2	1	1	100	100 Hz	1
Part 3	2	1	1	600	10 Hz	3
Part 4	2	1	1	1,000	1 Hz	0

Table 20 – Parameters of Test 6.

TEST ID	AVG DtU/s	STDDEV DtU/s
Part 1	97,60	4,07
Part 2	8,24	1,18
Part 3	9,96	0,12
Part 4	16,94	1,14

Table 21 – Test 6 results.

In the parameters table we introduced a new column “Total Disconnections” that shows the number of manual disconnections we did in the mobile device by switching off its wireless antenna. The goal was to evaluate the reliability and impact of intermittent connectivity in the MR-UDP protocol and data dissemination. In the raw test results we did not notice any DtU lost. Every disconnection lasts around 5 seconds.

In part 1 we did not force any disconnection because we were only interested to check if the middleware could disseminate the same 110 DtU/s to the mobile device. As the results table shows, the middleware can disseminate with the almost same rate as in Test 1. The average value is not 110 DtU/s, but the standard deviation is not very close to zero. It is also important to notice that the mobile device has less CPU and memory power and is communicating via a wireless technology (IEEE 802.11) where a large amount of collisions can happen because it is shared with many other mobile devices of the other users of the lab, which is a situation that does not occur if 3G/4G networks were used instead.

In parts 2 and 3 we manually caused disconnection periods. In part 3 a test round lasts for 10 seconds, so we only did one disconnection. In part 4 a round lasts 60 seconds, so we did 3 disconnections. In both cases none a single DtU was missed and after every reconnection all DtU were promptly delivered. It is important do notice that these disconnection was actually totally hidden to the applications and gateways by the MR-UDP protocol, for them it was like the connection was never lose. Moreover, the results show good dissemination values even in the **presence of intermittent connectivity**.

In part 4 both the DtP and the DtC was deployed in one mobile device. The value seems a little inconsistent from the previous test. However, we need to take into account that one mobile device was emulated (meaning that its performance was even lower) and that they both were sharing the same Wi-Fi network (with another number of phones and notebooks) and they both was generating traffic on the network in the same time.

6.2.7. Test 7 – The Last Dissemination Evaluation

For the last test of this section we would like to repeat the second part of test 3, where 50 different DtP produce DtU to an arbitrary number of DtC. This time one of the DtP was deployed on the mobile device (emulated) and an additional DtC was also deployed on another mobile device. As the reader can remember, in this test each MN has only one DtC to one kind of information randomly chosen. The Table 22 shows the test parameters and results.

TOTAL GW	TOTAL DtP	TOTAL DtC	TOTAL DtU	FREQ.	AVG DtU/s	STDDEV DtU/s
2	50	501	100	10 ms	90.99	13.58

Table 22 – Parameters of Test 7.

As the results table shows, the middleware provided consistent performance values even when mobile and fixed MN are used to produce and consume information.

Analyzing and combining the tests in all the previous sections, we can indicate that our model and middleware supports **scalable, reliable and resilient**

to intermittent connectivity on-time data dissemination to mixed mobile and fixed nodes with a dissemination rate near 3,300 DtU/s for each deployed Gateway.

6.3. Features Evaluation

Despite the fact that the ClientLib maintains an open connection from the mobile node to the Gateway, the MR-UDP protocol currently used is based on the connection-less UDP and promotes only a higher-level connection. With the native NodeConnection API and the extended GroupAPI and PubSubAPI, the ClientLib offers *decoupled communication* and an *extensible* API for implementation of new protocols by the developers. With the help of the MR-UDP features on the mobile connection, DDS on the core network and special services (e.g., MDT) the ContextNet offers *reliable communication* even in the occurrence of *handovers* and *intermittent connections*. The MR-UDP was implemented to cope to mobile connections, using *few resources*, *identifying IP changes* and able to *traverse Firewalls*. The Java implementation provides *heterogeneity*, since many platforms have virtual machines, including Android.

The high-performance core network supports the *on-time dissemination* of messages. The PoA-Manager, Group Definer and MTD Service are all optional, meaning that they can be deactivated if the applications do not need them. This makes the system *adaptable* to the context of the applications. However these services helps maintain a *scalable*, *expansible* and *dynamic adaptable* communication platform. For example, the PoA-Manager supports better use of resources, by requesting handovers from overloaded GW to more free ones. Moreover, it could deploy new GW if the current number does not meet the demand. The Group Definer also separates various communication channels, which helps in scalability. The MTD increases message *reliability*.

The SDDS provides the Data Management features. Offering abstract classes to Data Providers makes the system *expansible* since the developers can implement new providers at any time.

7 Conclusion and Future Work

In this thesis we proposed a data management model that enables deployment of a network of Data Provider components with reliable and on-time dissemination and transformation of information among many mobile nodes interconnected through wireless internet. We also presented the design of a middleware that implements this model and showed performance results that indicate that our model scales to thousands of mobile nodes and supports reliable, high throughput and on-time dissemination of messages between several thousands of mobile Data Providers and Data Consumers.

The main contributions of this thesis are the following:

1. The desing and implementation of a Mobile Reliable UDP protocol for mobile nodes which transparently handles short-lived temporary mobile node disconnections and ensures reliable packet delivery across these intermittent disconnections;
2. A communication middleware model and implementation, give evidence of its scalability, and show how it supports efficient and reliable unicast, *groupcast* and broadcast message delivery to mobile nodes in spite of IP address changes, temporary disconnections, and Firewall/NAT traversal by using our MR-UDP protocol;
3. A means for middleware and application developers to easily expand their models and application by adding a robust mobile communication via MR-UDP or new higher-level and application-oriented communication protocols, which will benefit from all features of our underlying model of SDDL/SDDS;

Results of several performance tests done in LAN and WAN settings show the suitability of our middleware for data dissemination and communications in large-scale mobile applications with thousands of nodes. Thus, we believe that our work offers a robust solution for the development and deployment of large-scale mobile applications that need reliable and timely dissemination of messages.

Despite these encouraging results, in future more stress tests are required, especially to find the performance limits of the SDDL core network. In particular, it would be important to measure how many DtU/s the core network using DDS can handle, independently of the number of DtPs. With the current tests we found that a single Gateway can handle around 3,300 DtU/s, but how many Gateways disseminating in this same rate can the SDDL core and the DDS domain handle? Moreover, the core network is not used only for dissemination of DtUs but for application messages as well.

We already have some QoS features in ClientLib/MR-UDP, like the reliable and timeless delivery of messages. However it is important to investigate how well other QoS features could be supported in this mobile distributed environment with wireless connectivity. For example, it should be fairly easy to add message durability to the SDDL, meaning that any newly added subscriber for information could be informed of previous published information. On the other hand, other QoS policies such as DDS' PriorityLanes or LatencyBudget (i.e. a time limit to deliver data) are much more difficult, if not impossible, to support over WiFi and 3G wireless connections.

Our current implementation only supports Java and Android, but there are several other platforms that do not run our middleware. Thus, it is worth the effort to make ClientLib interoperable between different systems. One possibility would be to use ContextML [45] (which uses XML) to make both the communication and the DtP definitions independent of the platform. The MR-UDP version described in this thesis uses Java object serialization, but a newer version already uses Protocol Buffers so as to achieve independence of the programming language at the communication endpoints. We also like the idea of using HTTP Rest, for the mobile communication and think that it could be intersting to investigate the benefits that this technology could bring for our model.

Moreover, the SDDS uses the Java ClassLoader to load and activate the DtPs. But we think we can investigate how to use component management middleware, like OSGi, and evaluate how we can benefit its services and only focus on our model evolution and implementation.

Taking advantage of the flexible deployment of Data Providers, the next logical step would be to provide a library of DtPs that support polling from different types of mobile device sensors, or else, implement common aggregation, transformation, or summarization functions for context data. Such a DtP library would certainly ease even more the development of context-awareness applications using ContextNet. Additionally, in a similar manner, we think it is important to investigate how we could provide means for unique context information naming to avoid conflicts and wrong notification of DtU with he same name.

The PubSubAPI already offers publish-subscribe communication with good performance, but it is important to evaluate in more depth the performance of the current filtering mechanism and investigate how it can support, for example, complex filtering expressions. Moreover, one should investigate if it is best to do filtering at the publisher or at the subscriber end. If done at the source, for example, it may be overloaded with many filters and the evaluation of each of these filters. On the other hand, if filtering is at the destination, the network can be overloaded with DtUs that may later be discarded anyway.

Finally, current design and implementation of the context model focuses on the deployment of DtP in smartphone-like devices. However, we have an increasing number of sensors embedded in many different devices and appliances, which use many different communication technologies. Thus, an interesting line of future work is to investigate how our model can be adapted to incorporate also such small sensor devices.

8
References

1. Dey, A.K., Salber, D., Abowd, G.D.: A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. Human-Computer Interaction 16 (2001) 97–166
2. Chen, G., Li, M., Kotz, D.: Design and implementation of a large-scale context fusion network. In: 1st Annual International Conference on Mobile and Ubiquitous Systems (MobiQuitous), IEEE Computer Society (2004) 246–255
3. Román, M., Hess, C., Cerqueira, R., Ranganathan, A., Campbell, R.H., Nahrst- edt, K.: Gaia: A middleware infrastructure for active spaces. IEEE Pervasive Computing, Special Issue on Wearable Computing 1 (2002) 74–83
4. Yau, S.S., Huang, D., Gong, H., Seth, S.: Development and runtime support for situation-aware application software in ubiquitous computing environments. In: 28th Annual International Computer Software and Application Conference (COMPSAC), Hong Kong (2004) 452–457
5. A. Dey, G. A. (Abril de 2000). Towards a Better Understanding of Context and Context-Awareness. Workshop on the what, who, where, when and how of context-awareness at CHI 2000.
6. TIBCO Inc. TIB/Rendezvous. http://www.tibco.com/products/rv/index.html.
7. Vitria BusinessWare. http: //www.vitria.com/products/businessware.html.
8. D. Stojanovic, B. Predic;, I. Antolovic et al., “Web information system for transport telematics and fleet management,” 9th International Conference on Telecommunication in Modern Satellite, Cable, and Broadcasting Services, (TELSIKS '09), pp. 314 - 317, October, 2009.
9. Carzaniga, A., Rosenblum, D.S., Wolf, A.L.: "Design and evaluation of a wide-area event notification service". ACM Transactions on Computer Systems 19(3), 332–383 (2001)
10. José Santa, Antonio F. Gómez-Skarmeta, Marc Sánchez-Artigas, Architecture and evaluation of a unified V2V and V2I communication system based on cellular networks, Computer Communications, Volume 31, Issue 12, 30 July 2008, Pages 2850-2861
11. Fei Ye; Adams, M.; Roy, S., "V2V Wireless Communication Protocol for Rear-End Collision Avoidance on Highways," Communications Workshops, 2008.

ICC Workshops '08. IEEE International Conference on , vol., no., pp.375,379, 19-23 May 2008

12. Jerbi, M.; Marlier, P.; Senouci, S.M., "Experimental Assessment of V2V and I2V Communications," Mobile Adhoc and Sensor Systems, 2007. MASS 2007. IEEE Internatonal Conference on , vol., no., pp.1,6, 8-11 Oct. 2007
13. Matolak, D.W.; Sen, I.; Wenhui Xiong, "Channel Modeling for V2V Communications," Mobile and Ubiquitous Systems: Networking & Services, 2006 Third Annual International Conference on , vol., no., pp.1,7, July 2006
14. Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. 2003. The many faces of publish/subscribe. ACM Comput. Surv. 35, 2 (June 2003), 114-131.
15. Oki, B., Pfluegel, M., Siegel, A., Skeen, D.: The information bus - an architecture for extensive distributed systems. In: Proceedings of the 1993 ACM Symposium on Operating Systems Principles. (December 1993)
16. Campailla,A.,Chaki,S.,Clarke,E.M.,Jha,S.,Veith,H.:Efficient filtering in publish-subscribe systems using binary decision diagrams. In: Proceedings of The International Conference on Software Engineering. (2001) 443–452
17. Banavar, G.; Chandra, T.; Mukherjee, B.; Nagarajarao, J.; Strom, R.E.; Sturman, D.C., "An efficient multicast protocol for content-based publish-subscribe systems," Distributed Computing Systems, 1999. Proceedings. 19th IEEE International Conference on , vol., no., pp.262,272, 1999
18. Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. 2000. Achieving scalability and expressiveness in an Internet-scale event notification service. In Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing (PODC '00). ACM, New York, NY, USA, 219-227.
19. Cugola, G.; Di Nitto, Elisabetta; Fuggetta, A., "The JEDI event-based infrastructure and its application to the development of the OPSS WFMS," Software Engineering, IEEE Transactions on , vol.27, no.9, pp.827,850, Sep 2001
20. OMG, "Data Distribution Service for Real-time Systems Specifications". www.omg.org/technology/documents/dds_spec_catalog.htm (visited on May. 5, 2014)
21. RTI. “RTI Connext — Comprehensive Summary of QoS Policies”. RTI, 2011. http://community.rti.com/rti-doc/500/ndds.5.0.0/doc/pdf/RTI_CoreLibrariesAndUtilities_QoS_Reference_Guide.pdf (visited on May. 5, 2014)

22. Ming Xiong ; Jeff Parsons ; James Edmondson ; Hieu Nguyen ; Douglas Schmidt; Evaluating technologies for tactical information management in net-centric systems. Proc. SPIE 6578, Defense Transformation and Net-Centric Systems 2007, 65780A (May 01, 2007)
23. G. Pardo-Castellote, "DDS Spec Outfits Publish-Subscribe Technology for GIG," COTS Journal, April 2005.
24. David et al.: A DDS-based middleware for scalable tracking, communication and collaboration of mobile nodes. Journal of Internet Services and Applications 2013 4:16.
25. Corradi, A.; Foschini, L.; Nardelli, L., "A DDS-compliant infrastructure for fault-tolerant and scalable data dissemination," Computers and Communications (ISCC), 2010 IEEE Symposium on , vol., no., pp.489,495, 22-25 June 2010
26. Y. Lee, S. S. Iyengar, C. Min, Y. Ju, S. Kang, T. Park, J. Lee, Y. Rhee, and J. Song. Mobicon: a mobile context-monitoring platform. Commun. ACM, 55(3):54–65, Mar. 2012.
27. C. Bettini, O. Brdiczka, K. Henricksen, J. Indulska, D. Nicklas, A. Ranganathan, and D. Riboni. A survey of context modelling and reasoning techniques. Pervasive Mob. Comput., 6(2):161–180, Apr. 2010.
28. Michael Knappmeyer, Nigel Baker, Saad Liaquat, and Ralf Tönjes. 2009. A context provisioning framework to support pervasive and ubiquitous applications. In Proceedings of the 4th European conference on Smart sensing and context (EuroSSC'09), Payam Barnaghi, Klaus Moessner, Mirko Presser, and Stefan Meissner (Eds.). Springer-Verlag, Berlin, Heidelberg, 93-106.
29. Yau, S.S.; Karim, F.; Yu Wang; Bin Wang; Gupta, S. K S, "Reconfigurable context-sensitive middleware for pervasive computing," Pervasive Computing, IEEE , vol.1, no.3, pp.33,40, July-Sept. 2002
30. Paolo Bellavista, Antonio Corradi, Mario Fanelli, and Luca Foschini. 2012. A survey of context data distribution for mobile ubiquitous systems. ACM Comput. Surv. 44, 4, Article 24 (September 2012)
31. BALDAUF, M., DUSTDAR, S., AND ROSENBERG, F. 2007. A survey on context-aware systems. Int. J. Ad Hoc Ubiq- uitous Comput. 2, 4, 263–277.
32. GADDAH, A. AND KUNZ, T. 2003. A survey of middleware paradigms for mobile computing. Tech. rep. SCE-03-16, Dept. of Systems and Computing Engineering, Carleton University, Ottawa.
33. HIGHTOWER, J. AND BORIELLO, G. 2001. A survey and taxonomy of location systems for ubiquitous computing. IEEE Comput. 34, 8, 57–66.

34. KJÆR, K. E. 2007. A survey of context-aware middleware. In Proceedings of the 25th Conference on IASTED International Multi-Conference: Software Engineering. 148–155.
35. VAN SINDEREN, M. J., VAN HALTEREN, A. T, WEGDAM, M., MEEUWISSEN, H. B., AND EERTINK, E. H. 2006. Supporting context-aware mobile applications: An infrastructure approach. IEEE Commun. Mag. 44, 9, 96–104.
36. Karen Henricksen, Jadwiga Indulska, Ted McFadden, and Sasitharan Balasubramaniam. 2005. Middleware for distributed context-aware systems. In Proceedings of the 2005 Confederated international conference on On the Move to Meaningful Internet Systems - Volume >Part I (OTM'05), Robert Meersman and Zahir Tari (Eds.), Vol. >Part I. Springer-Verlag, Berlin, Heidelberg, 846-863.
37. Segall, B., Arnold, D., Boot, J., Henderson, M., Phelps, T.: Content based routing with Elvin4. In: AUUG2K Conference, Canberra (2000)
38. Grossmann, M.; Bauer, M.; Honle, N.; Kappeler, U.-P.; Nicklas, Daniela; Schwarz, T., "Efficiently Managing Context Information for Large-Scale Scenarios," Pervasive Computing and Communications, 2005. PerCom 2005. Third IEEE International Conference on , vol., no., pp.331,340, 8-12 March 2005
39. Guanling Chen; Ming Li; Kotz, D., "Design and implementation of a large-scale context fusion network," Mobile and Ubiquitous Systems: Networking and Services, 2004. MOBIQUITOUS 2004. The First Annual International Conference on , vol., no., pp.246,255, 22-26 Aug. 2004
40. Guanling Chen, Ming Li, David Kotz, Data-centric middleware for context-aware pervasive computing, Pervasive and Mobile Computing, Volume 4, Issue 2, April 2008, Pages 216-253
41. Antony Rowstron, Peter Druschel, Pastry: Scalable, decentralized object location, and routing for large- scale peer-to-peer systems, in: Proceedings of the 2001 International Middleware Conference, Heidelberg, Germany, November 2001, pp. 329–350.
42. Martin Strohbach, Martin Bauer, Ernoe Kovacs, Claudia Villalonga, and Nils Richter. 2007. Context sessions: a novel approach for scalable context management in NGN networks. In Proceedings of the 2007 Workshop on Middleware for next-generation converged networks and applications (MNCNA '07). ACM, New York, NY, USA, , Article 5 , 6 pages.

43. Nguyen, T.H.; Sadiku, M.N.O., "Next generation networks," Potentials, IEEE , vol.21, no.2, pp.6,8, Apr/May 2002
44. Dowden, D. C., Gitlin, R. D. and Martin, R. L. (1998), Next-generation networks. Bell Labs Tech. J., 3: 3–14.
45. Moltchanov, B., et al.: Context-Aware Content Sharing and Casting. In: ICIN 2008, Bor- deaux, France (2008)
46. Kiani, S.L.; Knappmeyery, M.; Baker, N.; Moltchanov, B., "A Federated Broker Architecture for Large Scale Context Dissemination," Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on , vol., no., pp.2964,2969, June 29 2010-July 1 2010
47. Saad Liaquat Kiani, Ashiq Anjum, Michael Knappmeyer, Nik Bessis, Nikolaos Antonopoulos, Federated broker system for pervasive context provisioning, Journal of Systems and Software, Volume 86, Issue 4, April 2013, Pages 1107-1123
48. Marcio E. F. Maia, Andre Fonteles, Benedito Neto, Romulo Gadelha, Windson Viana, and Rossana M. C. Andrade. 2013. LOCCAM - loosely coupled context acquisition middleware. In Proceedings of the 28th Annual ACM Symposium on Applied Computing (SAC '13). ACM, New York, NY, USA, 534-541.
49. N. Carriero and D. Gelernter. Linda in context. Commun. ACM, 32(4):444–458, Apr. 1989.
50. Yasar, A.; Vanrompay, Y.; Preuveneers, D.; Berbers, Y., "Optimizing information dissemination in large scale mobile peer-to-peer networks using context-based grouping," Intelligent Transportation Systems (ITSC), 2010 13th International IEEE Conference on , vol., no., pp.1065,1071, 19-22 Sept. 2010
51. C. Esposito, S. Russo, and D. Di Crescenzo, “Performance assessment of OMG compliant data distribution middleware,” in 2008 IEEE International Symposium on Parallel and Dis- tributed Processing, 2008, pp. 1-8.
52. M. Xiong, J. Parsons, and J. Edmondson, “Evaluating the Performance of Pub- lish/Subscribe Platforms for Information Management in Distributed Real-time and Em- bedded Systems,” . omgwiki. org/dds, 2010.
53. OMG, “Data Distribution Service for Real-time Systems.” 2007.
54. K.-J. Kwon et al., “DDSS: A Communication Middle-ware based on the DDS for Mobile and Pervasive Systems”, Int. Conf. on Advanced Communication Tech-nology (ICACT08), 2008.

55. Esposito, C.: Data Distribution Service (DDS) Limitations for Data Dissemination w.r.t. Large-scale Complex Critical Infrastructures (LCCI). Mobilab Technical Report (March 2011)
56. Bova, T., and T. Krivoruchka. "Reliable UDP protocol." draft-ietf-sigtran-reliable-udp-00. txt (1999).
57. L.D. Silva, M. Endler, M. Roriz, MR-UDP: Yet another Reliable User Datagram Protocol, now for Mobile Nodes. Technical Report. 2013
58. Pardo-Castellote, G, Farabaugh, B., Warren, R.: "An Introduction to DDS and Data-Centric Communications." Real-Time Innovations. August 2005.