



HAGGLE 027918

*An innovative Paradigm for Autonomic
Opportunistic Communication*

Specification of the ADULT-Haggle

Deliverable Number: D1.4
Delivery Date: July 2009
Classification: Public
Authors: Haggle partners (Luca Della Toffola, Franca Delmastro, Silvia Giordano, Melek Önen, Andrea Passarella, Daniele Puccinelli (editor), Christian Rohner, Abdullatif Shikfa, Salvatore Vanini)
Document Version: v1.0 (September 29, 2009)

Contract Start Date: 1 January 2006
Duration: 4 years
Project Coordinator: Thomson, France
Partners: EPFL (Switzerland),
Uppsala University (Sweden),
University of Cambridge (UK),
Martel (Switzerland),
Eurecom (France),
CNR (Italy),
LG (France),
SUPSI (Switzerland)

**Project funded by the
European Commission under the
Information Society Technologies
Programme of the 6th Framework
PRIORITY 2
(2002-2006)**



Abstract

The Hagggle node architecture evolution follows an incremental approach, passing through different stages, each of which involves a different set of functions, capabilities, and knowledge. In this deliverable we present the ADULT-Hagggle architecture, which completes the evolutionary process started by refining the YOUNG-Hagggle specification. After summarizing the main considerations that drove the design of the previous specifications, we focus on the evaluation of the current Hagggle implementation in order to validate its mechanisms. Moreover, we describe the design of the ADULT-Hagggle architecture and the refinements made in the areas of resource management and security.

Contents

1	Introduction	5
1.1	A brief history of Haggle	5
2	Experimental evaluation	7
2.1	Power Consumption	7
2.2	Live Experiment	8
2.2.1	Delivery Fraction	9
2.2.2	Traffic Pattern	10
2.2.3	Hop-count and Delay	10
2.3	Lessons learned from experiments	11
3	Architecture refinements	12
3.1	Resource-sharing communities	12
3.1.1	Distributed resource sharing in a community	12
3.1.2	Solutions for resource sharing	14
3.1.3	Framework	15
3.2	Social-aware content sharing	23
3.2.1	Context definition and cooperative downloads in opportunistic networks	24
3.2.2	Application scenario	26
3.2.3	Integration within the Haggle architecture	27
3.3	Security	28
3.3.1	Encrypted Haggle Certificates	29
3.3.2	Definition of the setup phase	29
3.3.3	Securing the communication channel	30
4	Conclusions	31

List of Figures

1	Battery lifetime of the HTC Touch Diamond.	8
2	Fraction of data objects received over time.	9
3	Distribution of inter data object times.	10
4	Hop distance and end-to-end delay.	11
5	Interface sharing	19
6	Example of cooperative download	27
7	Context-aware data dissemination module	28
8	Screenshots of the social-aware content sharing service GUI	28

1 Introduction

This deliverable completes the incremental model of the development of the Haggle node architecture, which started with a basic INFANT-Haggle that performs simple autonomic activities, and now achieves its maturity as ADULT-Haggle. Before presenting the specification of the ADULT-Haggle architecture, we give a short description of the architectural choices (and their motivations) and the decisions we undertook during the evolution phases of Haggle.

1.1 A brief history of Haggle

The first stage of the Haggle node architecture was INFANT-Haggle, which was characterized by a minimal set of functions, capabilities, and knowledge. To face the complexity, dynamism, and uncertainty of a network environment featuring intermittent connectivity and opportunistic communications, we identified a set of key principles in INFANT-Haggle that have guided the design of the architecture throughout its later stages (see deliverable D1.3).

In INFANT-Haggle, we made a key decision that was maintained in the later stages of Haggle: the adoption of a modular structure consisting of managers. A manager can be seen as an autonomous agent and is an entity with a well-defined role and responsibility within its local domain. The manager-based structure has proven to increase the flexibility of the architecture.

In the second stage of Haggle, known as CHILD-Haggle, we introduced an event-based interaction model among managers to solve the performance and scalability flows discovered during the first experiments on the early Java implementation of the INFANT-Haggle node architecture.

In our event-based scheme, each manager produces and consumes a set of public events and is oblivious to the event generator(s), making it possible to easily add/remove managers to/from the architecture. Moreover, to provide a more sophisticated communication model to the CHILD-Haggle architecture, we introduced novel and efficient forwarding algorithms. To empower and streamline information management, we added some initial autonomic features that exploited the mobility of the node and a data management framework. Finally, to integrate security in the CHILD-Haggle architecture, we proposed a preliminary design of a security manager.

In the YOUNG-Haggle specification, we learned from previous trials and implementation work how to refine certain aspects of the CHILD-Haggle architecture, as well as introduce new concepts.

To facilitate the search aspects of Haggle and its initial goal to perform "ad hoc Googling", we introduced the concept of a search based network architecture, which deals with the set of search primitives embedded in the architecture to support Haggle's data centric communication approach.

To manage resources such as energy, storage and computing, we specified a new resource management system whereby, a resource manager publishes policies, and the implementation of these policies is then delegated to individual managers, which adapt their operations to the issued policies.

In order to complement infrastructure networks and exploit them based on their availability, we defined a communication framework with non-Haggle aware computers over the Internet,

which also includes the integration of legacy applications into Hagggle.

Finally, we defined a security framework for Hagggle which enables secure community-based communication. This proves a node is a member of a Hagggle community and ensures the privacy of context information which is then exchanged in order to obtain forwarding decisions.

2 Experimental evaluation

In this section we provide an experimental evaluation of the Haggle implementation to validate its effectiveness in supporting opportunistic communication on the basis of the remarks described in section 2 of deliverable D1.3. Our objective is to exploit the feedback received from the experiments to refine the YOUNG-Haggle architecture and then define the ADULT-Haggle specification.

In order to accomplish our objective, we evaluate the following: 1)the capability to capture contact opportunities; 2)the capability to deliver data to destinations; 3)the compliance of search based resolution with intermittent connectivity; and 4)the utility of forwarding data over multiple hops. The scenario we used is a typical office setting, where a laptop is generating Garfield comic strips and is disseminating them to seven mobile phones using Bluetooth. The phones run the PhotoShare application (see deliverable D3.3) and are carried by research colleagues during a day of normal office activities. These activities include meetings, office work, and lunch outside their offices. Each participant has entered the attribute `Picture=garfield` into PhotoShare to express an interest in the Garfield strips, which they can also view when it is received. Each time a phone encounters another phone, or the laptop, they both exchange node descriptions and perform search-based resolutions to determine if they have any comic strips to further exchange. Since every participant has entered the same interest, the strips should ideally spread to all mobile phones either directly from the source, or via continuous resolutions over the phones. We limited resolutions to include only the first ten results (i.e., strips) that were not yet received by a peer. Thus, a node will never receive more than ten strips at a time.

2.1 Power Consumption

A limiting factor in our experiment is the battery lifetime of the mobile phones. We used three HTC Touch Diamonds and four HTC S-620 that operate on Windows Mobile. The Diamond is a more advanced phone than the S-620, but its battery lifetime is also significantly shorter, lasting only the duration of the live experiment (the S-620 lasted more than 24 hours).

The laptop we use is an IBM Thinkpad X31 with Ubuntu 8, internal Intel WiFi, and external Bluetooth.

The first aspect we hence examined is Haggle's effect on a mobile phone's power consumption. We benchmark a single HTC Touch Diamond mobile phone using both Bluetooth and WiFi (WiFi in the live experiment was not used, as described in section 2.2. There are three power mode settings for WiFi on the Diamond: best battery (BB), best performance (BP), and auto mode. Auto mode adjusts the power against signal quality, and for coherent results, we show only BB and BP. Bluetooth only has one mode of operation.

As a baseline, we ran a phone isolated with only neighbour discovery on either Bluetooth or WiFi. With Bluetooth, Haggle performs a device scan every 60 ± 45 seconds and with WiFi it sends a broadcast beacon every 5 seconds. This provided us an estimate of the impact of neighbour detection. In order to compare with the baseline, we added the laptop, which generates a new 30-60 KB comic strip every minute. The Garfield strip is then transferred to the phone closest to the laptop. The application data rate is quite high for this scenario, and delivered an

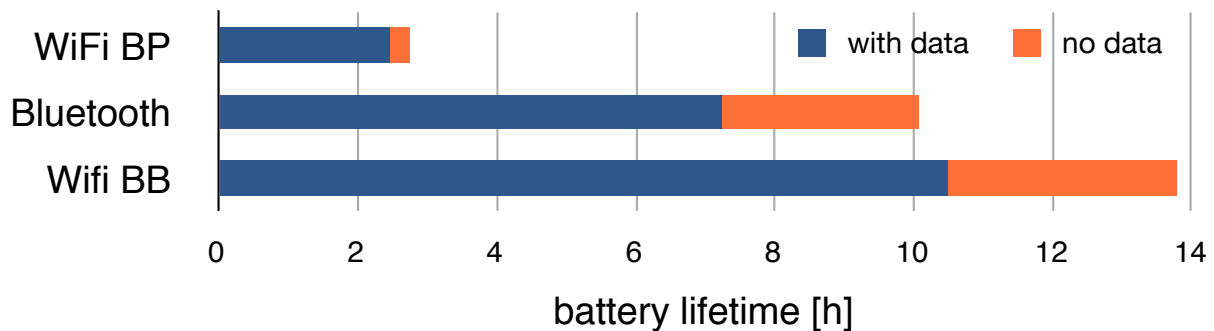


Figure 1: Battery lifetime of the HTC Touch Diamond.

estimate of data traffic impact in a busy environment.

Figure 1 demonstrates the results from the power benchmarks. The most striking result is the short battery lifetime of WiFi BP. The 2-3 hours of running time is clearly too short for any realistic network scenario. WiFi BB, on the other hand, has the best lifetime of all modes, including Bluetooth. However, the BB mode operates at very low power output and the mobile phone and laptop have to be placed very close to each other for reliable data transfer. This limits the usage of BB mode in practice, but in combination with BP it effectively shows the upper and lower bound of auto mode.

Bluetooth costs more in terms of neighbour detection compared to WiFi BB. On the other hand, we actually found Bluetooth to be more useful for data transfer due to its increased range over the WiFi BB mode. Neighbour detection with Bluetooth is not as reliable as WiFi, because a device cannot be detected while scanning. Therefore, the scan collisions increase with the density of the network. We found that Bluetooth provides a reasonable trade-off between device longevity and the service provided, lasting up to seven hours with data. This is enough to last a normal working day, without frequent recharging. Bluetooth is therefore our technology of choice for prolonged experiments.

2.2 Live Experiment

For the live experiment in mobility measurements, we lowered the intensity of the comic strips to 10 minutes intervals and increased the Bluetooth scan interval to 80 ± 60 seconds. We chose this longer interval because devices were less likely to scan simultaneously when we had several phones.

In terms of battery lifetime, our reference phone managed 6.5 hours, which is slightly shorter than the static setup with data. Although we use lower data and scan rates, the live experiment showed frequent neighbourhood changes that caused failed transfers and hence retransmissions. We also increased the cost of neighbour detection by using several phones since the phones must also respond to scans.

During previous experiments we noticed a problem with the Windows Mobile Bluetooth stack since it sometimes goes down and resets to OFF mode. We therefore instructed our colleagues to be observant of any changes in the Bluetooth settings, and to turn Bluetooth back on if

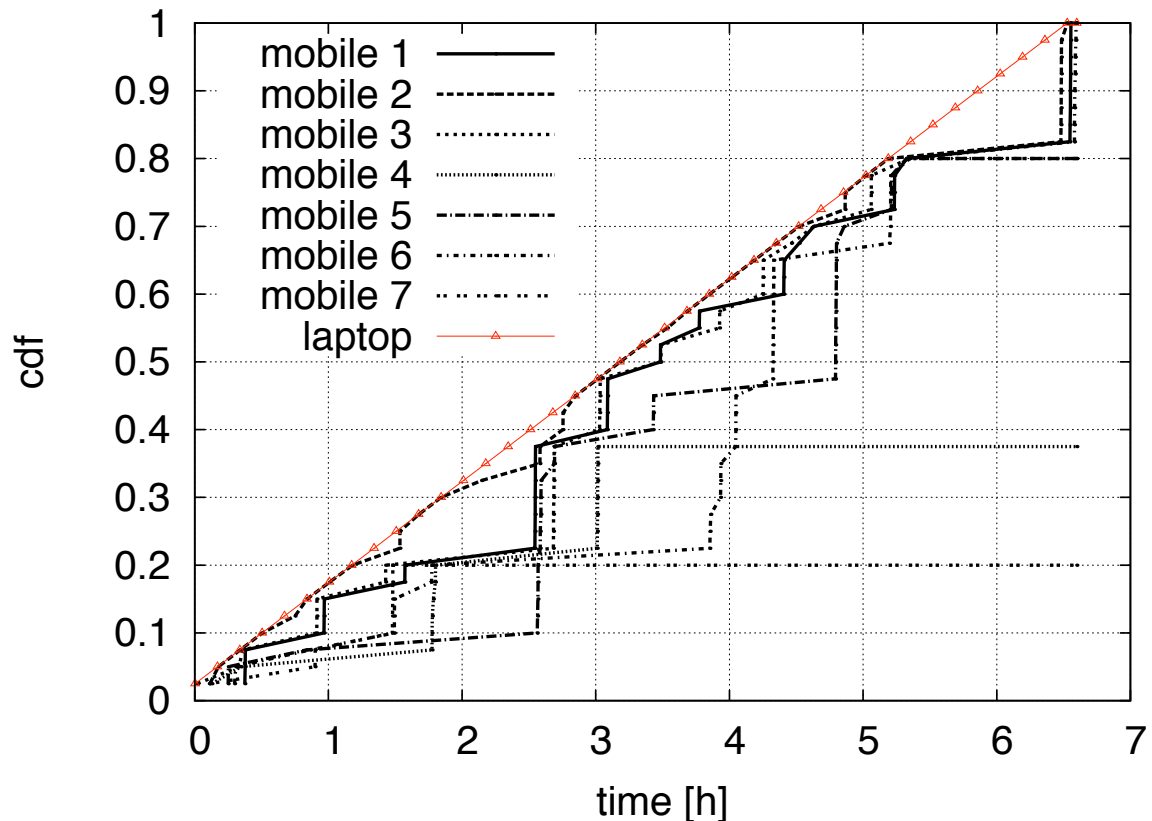


Figure 2: Fraction of data objects received over time.

found in OFF mode. However, sometimes the phones may still indicate ON mode after the stack goes down and therefore, becomes difficult to detect. The only solution in this case is to turn Bluetooth off and on again. This problem had an effect on our results in that phones on rare occasions miss contact opportunities or, in worst case, are unconnected for prolonged periods of time.

2.2.1 Delivery Fraction

In Figure 2, we see the fraction of received data objects over time, up until 6.5 hours when the experiment ended. The laptop serves as the reference point as it gets the data objects directly from the application that generates them. Ideally, all phones should have the same delivery fraction as the laptop. The figure shows the different connectivity of the mobile phones. For example, mobile 2 is a phone that was co-located in the same room as the laptop for the most part of the experiment. Mobile 7, on the other hand, was isolated in another office and sees very little progress (only at the beginning of the experiment). Three phones received all data objects by the end of the experiment. Mobile 7 and mobile 4 only received 20% and 30% of the data objects, respectively. However, we do not believe isolation is the only explanation for these low delivery fractions, but we suspect the above mentioned Bluetooth problems also play

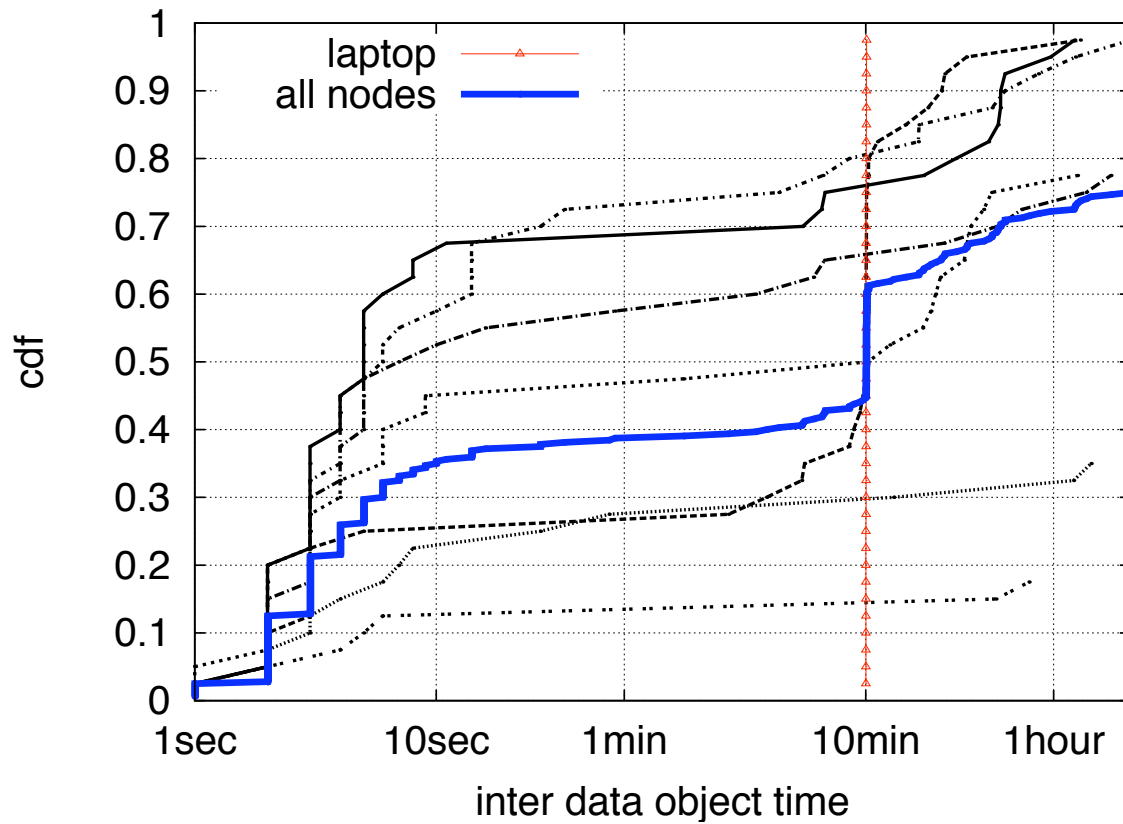


Figure 3: Distribution of inter data object times.

a role.

2.2.2 Traffic Pattern

Figure 3 shows the distribution of inter data object receive times. This source has the expected constant 10 minute interval. Most of the mobile phones receive around 50% or more of their data objects within an interval of less than 10 seconds. This shows that many data objects are sent and received in bursts when a phone co-locates a new neighbour. With search-based resolution, the cost of a resolution is independent of the number of data objects resolved (it is only related to the size of the relation graph). Therefore, we contend that search-based resolution fits the intermittent connectivity of opportunistic networks successfully where bursty traffic is commonly found.

2.2.3 Hop-count and Delay

A significant question is how often a mobile phone gets its data objects from the laptop, or from another mobile phone that forwards the data objects. The answer is found in Figure 4 (left), where the data objects hop count distribution is shown (we measured the hop count once

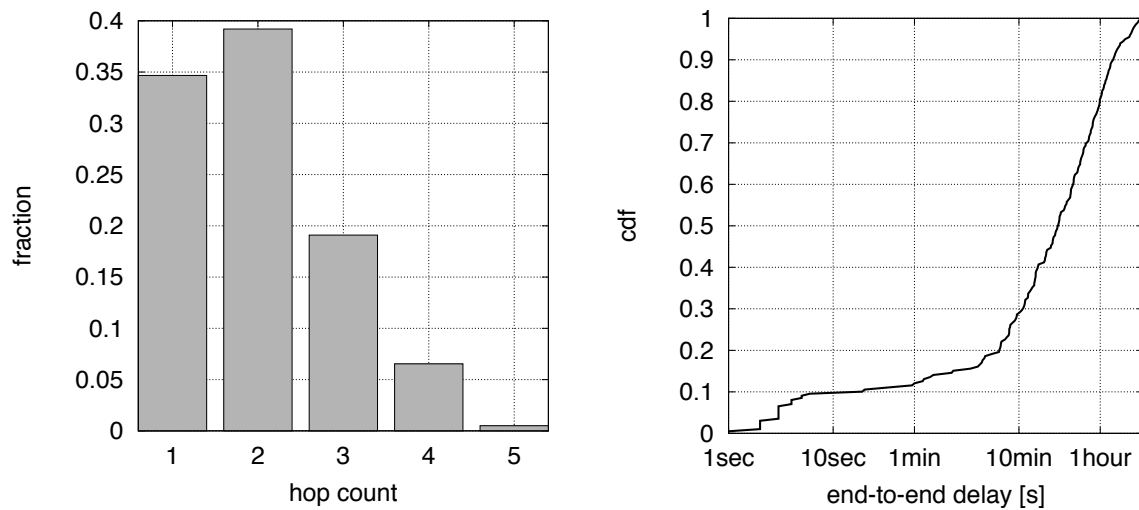


Figure 4: Hop distance and end-to-end delay.

for each receiver of the same data object). 65% of the received data objects are forwarded over multiple hops, showing that Haggle provides a multihop dissemination service for the majority of data. In terms of delay (Figure 4 right), 30% of the received data objects are delivered within 10 minutes, while the rest are delivered within 2 hours. From these results we conclude that Haggle’s continuous resolution system works successfully for providing basic interest forwarding.

2.3 Lessons learned from experiments

The conclusion we can draw from the power consumption (static and live) experiments is that mechanisms for managing network interfaces save energy but need further refinement.

From the live experiments we can conclude that Haggle’s search-based resolution mechanism is effective for the intermittent connectivity of opportunistic networks, both in terms of number of data delivered to the destination and the time occurred for data to be delivered.

An important question that arose is how to optimize data objects sharing among Haggle nodes: in section 3.2 we demonstrate how this is possible by exploiting context and content information.

3 Architecture refinements

In this section we introduce the main refinements of the YOUNG-Haggle architecture that have lead to the specification of the ADULT-Haggle.

The main contributions include the following: a framework for sharing resources (processor cycles, storage capacity, network interfaces) among members of a community which can be useful in case of limited resources; new Haggle security features that allow a node not to reveal an attribute to any other node; and the integration of the data dissemination framework presented in deliverables D1.2 and D1.3 into the Haggle architecture.

3.1 Resource-sharing communities

In recent years there has been a development of new trends and communication paradigms that are radically changing network architectures and protocols. Human and community-centric services are one of the most widely recognized driving forces, because network structures arise according to interpersonal social relations as well as individual needs and interests.

A great deal of work has been done in community detection within networks [13, 7] and in the exploitation of the social relationships among community members. For example, in deliverable D2.2 we demonstrated that the BUBBLE Rap forwarding algorithm is able to exploit social structures to greatly improve the forwarding efficiency compared to oblivious forwarding schemes.

In the following, we describe a framework for the sharing of resources (such as processor cycles, storage capacity, etc.) among the members of a community. Our framework and the protocols that are derived from it leverage from the key mechanisms in the Haggle architecture. Furthermore, the framework primitives can be considered as an extension of the Haggle's features.

3.1.1 Distributed resource sharing in a community

A community consists of a group of people sharing similar interests and performing similar activities. Such people engage in socially meaningful relationships where prominent patterns of information flow can be observed.

Community membership comes with three key benefits: cooperation (members can work together to achieve a common goal), support (members can receive help from their fellow members), and constructive peer pressure (to discourage undesired behaviour).

In this document, our focus is on cooperation. The Internet is a case in point: individual users form virtual groups that interact with just a few restrictions by exchanging data and messages. People within a group also share resources in the form of data, code, or processors using dedicated protocols.

A particularly widespread form of Internet resource sharing occurs in the distributed object-oriented paradigm, where sharing based on the client-server model of distributed objects [18] results in code reuse and aids the software development process. Storage sharing within an organization over the Internet has also been studied [14].

In a Pocket-Switched Network (PSN), there is no a priori information about the network struc-

ture, and communication relies on human mobility. People who comprise the network are tied by social relationships driven by the patterns of connectivity among individuals. Cooperation in PSNs is an important attribute that binds and also divides people into communities. The BUBBLE Rap forwarding algorithm leverages on the tendency of people to communicate with individuals within their own community and indeed works out a popularity ranking within a community: community information is exploited to select forwarding paths. Now, we wish to leverage on communities to enable resource sharing among members.

To achieve this goal, communities must first be discovered. In [11] we already examined some centralized community detection methods and investigated the possibility of developing a distributed version tailored for PSNs, where mobile devices can sense and detect their own local communities instead of relying on a centralized server. The two informal elements that make it possible to classify relationships among members of a group are familiarity and regularity, which can be quantified by the following metrics:

- Shared interests, which can be inferred based on user behaviour and context (if a person watches a TV program about cars at the same time and day, she/he is likely to be interested in cars.)
- Contact duration, which correlates with familiarity. Contacts are considered valid if they exceed a predetermined threshold.
- Contact distribution, which is a good indicator of the existence of a social relationship between people.
- Similarity, including structural equivalence (two nodes are equivalent if they share a common set of neighbours) and approximate equivalence (using Euclidean distance and Pearson correlation [13].)
- Number of paths between nodes, which can be viewed as a measure of the influence of individuals on other individuals within the community (the more paths that exist, the more opportunities one node has to affect the other.)

Shared interests and number of paths are closest to Haggle's concept of a relation graph, one which allows data objects to represent nodes and relationships among nodes in the network. The attributes in a node data object are the data interests of the node. In a relation graph, a relationship between nodes means that they share at least one common attribute in their metadata. We assert that the existence of at least one common interest is a necessary but not sufficient condition for two nodes to belong to the same community. A newly encountered node is considered a member of the local community if its cumulative contact duration time exceeds a given threshold. We rely on communities for implementing distributed resource sharing, and we exploit Haggle's relation graph to search for a set of common interests that identify the community.

Our goal is to build and identify communities so that their members can share resources such as communication links, memory, storage and processor cycles. This is particularly useful in situations such as:

- Limited resources. Nodes with scarce resources (*e.g.*, low battery level or low memory availability) can forward important data to members of the same community.

- Idle state. If a node is in an idle state (no tasks to carry out), it can offer its computational resources to members of its community.
- Resource economy. A node can use the computing resources of other members of its community to preserve its own resource for future use.

As in the real world, there can be an incentive for a node to join a community and offer its own resources, and members can gain a reputation through the ratings of their peers (as in the RentACoder web-based marketplace [1]).

Incentives can be used to encourage prospective members to join a community, to encourage members to stay in that community, and to encourage active participation. A community should continuously adapt their incentive plan to suit their changing needs.

In the following section, mechanisms known in the literature for sharing processor and excess storage capacity are described.

3.1.2 Solutions for resource sharing

Since our purpose is to exploit Hagggle along with community detection mechanisms to enable resource sharing, we begin with a survey of existing solutions for resource sharing.

The current work in resource sharing is mainly based on mobile agents and Remote Procedure Calls (RPC) (in this specific case, Hagggle's metadata management is being exploited).

3.1.2.1 Mobile agents *Mobile agents* are autonomous programs that can travel from computer to computer under their own control [15]. They can provide a convenient, efficient, and robust framework for implementing distributed applications including mobile applications. Mobile agents are the best theoretical solution for solving the resource sharing problem. Mobile agents are a well known topic but the technology is still not widespread because the current status of the technology is not appropriate for a mainstream diffusion [12].

Examples of the large body of literature devoted to mobile agents include [12] and [15]. Mobile agents are an elegant solution to inter-host code transfers. If resources are scarce, mobile agents allow tasks to be interrupted, transferred, and resumed in a different host. Mobile agents do not require a predefined common shared set of functions and provide total flexibility in development. Drawbacks include potential security breaches (*e.g.*, transfers of malicious code) and implementation difficulties. For instance, if the execution must be interrupted atomically at a precise point (*e.g.*, at a certain program counter value) and later resumed at the same point, the implementation and the technical issues would be difficult to solve [17], [8]. On the other hand, if a fine-grained precision is not required, simpler techniques like plain serialization or the adoption of scripting languages could be utilized.

3.1.2.2 RPC Communication Remote Procedure Calls (RPC) are an alternative to mobile agents. Classic RPC approaches like IDL, Corba, or XML-RPC allow calling functions that are remotely stored in another node. The main difference between this approach and mobile agents is that a node needs to expose a-priori list of functions or provide a mechanism (*e.g.* a query packet) to discover the node's capabilities. Since a node is typically unable to dynamically look

at a function list and infer which functions it needs to perform a task, the simplest solution is to provide a shared knowledge base of utility functions that each node can exploit from the others.

The RPC approach is implementation-independent. Each node stores its internal implementation that is hooked to the underlying platform and needs only to comply with the provided interfaces. RPC is more lightweight than mobile agents (in terms of consistency control and implementation effort) and is more energy-efficient, because it needs fewer data transfers.

3.1.2.3 Metadata Metadata is a variant of the RPC protocol implementation. Each RPC implementation needs to be compliant with the protocol specifications; this means that it is necessary to provide a custom implementation of the protocol or using existing libraries. In our case it is possible to use the existing Haggle mechanisms (such as data objects) as the medium to transfer data between nodes.

Exploiting the current Haggle implementation requires less effort and changes in the architecture to provide RPC-like services. By using Haggle, total freedom in adding and removing features to an existing protocol or even inventing a new one is achieved. The main drawback is that by creating a custom implementation, we do not exploit existing well known and tested libraries.

3.1.3 Framework

In this section we present our proposed framework for distributed resource sharing in communities. We describe the metrics for the evaluation of a node based on community opinion and we present community incentive schemes. In section 3.1.3.3 we describe a possible implementation in the current Haggle reference architecture.

3.1.3.1 Task assignment and Model As described in the introduction, one of the reasons why distributed resource sharing can be used is for energy conservation: jobs can be migrated from a mobile node to another for execution in order to reduce CPU energy consumption.

However, there are scenarios where a framework for distributed resource sharing is not convenient: for example, if an inter-host data transfer is more energy-costly than local data processing and storage, or the remote execution of a task on a node prevents the usage of its own resources. In these cases, resource sharing requires a value assessment of delegating a task to another node, and an algorithm to manage the transfer decision.

Usually, the approach taken for transferring a job's state, is to compare the cost for migrating (which comprises also the transfer time) combined with the local execution cost. Such costs can be assessed with an energy consumption model of the device in use or with empirical data (e.g., the energy consumption of WiFi or the CPU). In the first case, the better the model, the more accurate are the predictions and the obtained results, and consequently it will be possible to better distribute high consumption operations to more powerful devices. Since Haggle logs jobs execution time, we chose the empirical data method and adopted an adaptive algorithm, which learns and adapts its decision to migrate based on a job's execution history. The algorithm works as follows.

A job is always run locally for the first time, and its CPU running time is logged. After its

first run, a running tally of the average CPU time for that specific job is used to assess if the remote execution would be beneficial. This is done by comparing the cost of running a job locally (which is a function of CPU time and CPU power consumption) with the cost of migrating a job (which is a function of the size of the job to be transferred and the power required to receive/transmit packets). Given:

P_{CPU} = power consumed to execute a job on the mobile node (W)

P_{xmit} = power consumed to transmit a packet (W)

P_{rec} = power consumed to receive a packet (W)

S_{JOB} = size of the job to be transferred

T_{JOB} = CPU running time for a job on the mobile node (sec)

S_{res} = packet size of the result (fixed)

B = bandwidth of the network interface

Cost of executing a job on the mobile node = $P_{\text{CPU}} * T_{\text{JOB}}$

Cost of migrating a job = cost of transmitting a job + cost of receiving the result
 = $(S_{\text{JOB}}/B) * P_{\text{xmit}} + (S_{\text{res}}/B) * P_{\text{rec}}$

To fully exploit its potential, the computational resource sharing scheme can be used for both conditions of low remaining energy and during normal operation.

After evaluating whether a migration is in order, the second step is to determine which node of the community to migrate resources to. To this end, nodes exchange their *resource profile* indicating their available resources (e.g., physical memory availability). The resource profile offers no contextual information about resource usage. Different users can have different patterns of contribution, and therefore, an incentive scheme should be employed. Finally, to avoid the *ageing of a community* [16], characterized by a small number of users providing a large proportion of the contributions, it is advantageous to motivate users to contribute high-quality resources while simultaneously blocking the contribution of lesser-quality resources. To accomplish this end, a field was introduced in the resource profile called *Community/global rank* (a similar mechanism is presented in [10]).

Global rank is a metric that is used for a local node to evaluate a remote node based on its involvement in the resource sharing process. Nodes are ranked based on the total number of their contributions including: type of resource(s) shared, number of times it was shared, shared resources over total available resources, and, for communication links, the percentage of used bandwidth. The node's *reputation* within the community is quantified as the sum of all its ratings for all its contributions. The reputation of a node is updated after the completion of each resource sharing process to reflect the node's ability to share high-end resources. It is also used as a reward mechanism that consists in assigning a node a priority proportional to its reputation; the priority level determines, in case of multiples resource sharing requests (e.g. two nodes want to use the storage space of a neighbour at the same time), which node to assign resources to.

Our resource sharing framework works in parallel with the utility based data dissemination mechanism presented in deliverables D1.3 and D1.2. We recall that the scenario envisaged for

data dissemination is the one in which the Haggle nodes generate data objects that they wish to share with other interested nodes: to this end, a utility function allows the optimal use of local storage to achieve the best hit rate for all nodes. The objective of the resource sharing framework is to provide computational resources to nodes which have requested them and are from the same community. In this regard, data is forwarded and returned to nodes through a forwarding algorithm. The information contained in the resource profile can potentially be used by forwarding algorithms to determine the best delegate forwarder for data in transit.

3.1.3.2 Shared resources A list of shareable resources (indicated in the resource profile) is as follows:

Processor. Configuring the amount of CPU power to be shared was not included in this analysis since this value is subject to continuous fluctuations and cannot always be promptly updated in an opportunistic setting. Each node willing to share its processor provides all its computational power to the remote nodes when asked (*i.e.*, the operating system is responsible for the CPU management).

As described above, the framework lists the computational resources of a node in the resource profile. In this case, the information contained is the CPU speed (*e.g.* values in MHz, GHz), which also serves to estimate the global rank metric (that also includes additional information about the cache and the RAM). The global rank metric is used by other nodes to determine the operational profile of a given remote node and decide whether it is a good resource delegate.

In section 3.1.2 we presented a number of techniques that define how computing power can be exploited by the members of a community. Our approach to sharing the processor cycles is to add an additional abstraction level on the application side, which uses RPC style method calls (*e.g.* like in Java RMI or .NET Remoting [2]). The information transfer format is metadata. An application that is interested in resource sharing, or distributed execution, uses the framework to announce its intent. The application exposes the list of methods that can be executed on other nodes. Depending on the level of resources and the outcome of the migration cost analysis described previously, the framework transparently switches the execution context of the specified methods and delegates their execution to other nodes of the community. This process requires an initial handshake whereby a request packet is sent to the remote host and contains:

- The operation identifier (processor sharing request).
- An indication if other resources are needed (such as storage).

The remote node replies with a response packet, which can contain additional parameters. The context is transferred by sending packets using metadata whose format is very similar to the one used by the RPC-XML protocol. As was mentioned previously, we made this choice because in this way we can exploit the existing mechanism that allows the insertion of additional metadata into a data object exchanged between two peers. The requesting node supplies the method, the values of the parameters, and the necessary data to the remote host, using the format:

<Context>

```

    <item name="method" value="remoteMeth1" />
    <item name="param1" value="param1Value" />
    <item name="param2" value="param2Value" />
    <Data href="file:///data.dat" />
</Context>

```

Sent data may be stored in the remote node until processor resources become available, in which case the remote node needs to create a directory for storing guest files and the computation results. If the method to be executed needs to perform file I/O, during the initial handshake the remote node inserts the value of the maximum amount of storage allocated into the response packet.

Storage. Nodes belonging to a community that want to share storage need to specify how much storage space they are willing to offer to other members. For this reason the framework must provide the primitives that assist in the configuration and the maintenance of the shared space. The amount of storage space a node wishes to share is fixed and set a priori, whereas the amount of storage space that is remotely available is exposed in a specific field in the resource profile. Since this value becomes easily outdated, an initial handshake is performed before transmitting data, in order to verify if the host node has sufficient space to store data. Nodes that ask for storage space can borrow from the community up to a credit limit that is proportional to their global ranking. Nodes can employ distributed data sharing for the following reasons:

- Space is no longer available on the local data store.
- Multiple backups. This happens when sensible data should be replicated because of security reasons.
- A host needs to transfer its current status due the lack of resources (*i.e.* low battery level).

In all cases, the host has to remember where data/state is remotely saved, and there should be a mechanism that allows an application to reclaim storage from a node holding its data. In addition, the application applies a security policy regarding the pieces of data that it wants to store in a remote space. Therefore, we have defined three distinct security levels. Each level provides a different reward rate to the node that accepts hosting the data, and that is reflected in its global ranking. The higher the security level, the greater the node is rewarded based on the expensive effort of resource usage. The three security levels are as follows:

- **SECURE_SHARED:** data cannot be deleted.
- **TRUSTED_SHARED:** data is stored in different nodes. If a node wants to drop some data to free up its storage space, it must find another replacement delegate.
- **SHARED:** data is not securely locked. It can be deleted anytime and anywhere in the community.

The framework for sharing storage space within a community follows a different approach with respect to the context-aware content sharing schema described in deliverable

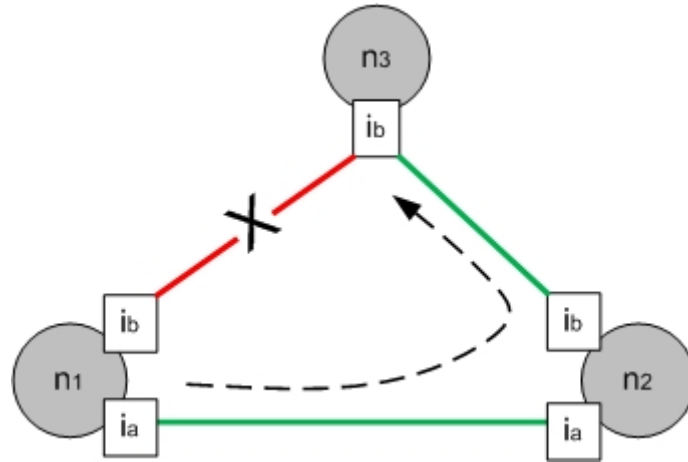


Figure 5: Interface sharing

D1.3. The framework allows a node to store data in another node due to the lacking of local storage space or for multiple backups reasons. Furthermore, each node decides a priori how much storage space it would offer to other members of the community and this value is fixed. In contrast, within the context-aware content sharing schema, each node, each node decides to use part of its storage space to store data that can be of use for nodes it will encounter in the future. This occurs independently from any explicit request coming from other nodes. To accomplish this task, it analyses its own context and those of its neighbours and previously encountered nodes. Furthermore, the amount of shared space is not fixed but determined by a utility function based on data objects stored by other peers and data objects it currently stores.

Interfaces. As for the processor sharing scenario, network interfaces are exploited for resource savings, and this is one of the directions we decided to investigate after the experimental evaluation of the Haggle implementation (see section 2). The benefits that can derive from network interfaces sharing are clarified by the following example. Let us assume that three nodes n_1, n_2, n_3 belong to the same community (see Figure 5). n_1 use interfaces i_a of type a and i_b of type b, with i_a being less power-hungry than i_b . n_2 employs interfaces i_a and i_b , while n_3 only uses i_b . At a certain location, n_1 and n_3 communicates through i_b . At a certain time, n_1 energy level becomes too low. The framework recognizes that n_1 has an inactive interface i_a that is less power-hungry and decides to turn off i_b , but n_3 can only count on i_b . The scheme determines that n_1 has another neighbour n_2 , through which it can reach n_3 , since n_2 has both interfaces i_a and i_b . The communication between n_1 and n_2 is then carried out via i_b because it is less costly. This mechanism allows n_1 to save energy but also to maintain a communication link with n_3 , although it requires that the devices remain visible between them simultaneously, multi-homed hosts and delay tolerant applications.

3.1.3.3 Framework implementation in Haggle The current implementation of the Haggle architecture offers a subset of the mechanisms required for implementing the resource sharing framework. Haggle code provides all the necessary function implementations and libraries

to manage metadata and data objects, specifically providing the security library denoted as "attribute certificate library". This allows for a secure communication which is community based. For the most part, the changes needed to implement the framework affect the application layer, because processor and storage sharing are triggered by applications. For this reason, all changes and additions should be made in *libhaggle*, which already implements a communication link with the Huggle daemon, as well as a set of utility functions to manipulate data objects.

The resource sharing framework requirements are scalability, ease of use, and transparency, in order to be easily integrated. Furthermore, the resource sharing framework should not impact the overall design of the application.

The resource sharing framework is in general useful when a node's resources become scarce. In this regard, the framework defines different levels of alerts (*e.g.* different levels of battery depletion) and an application can register for the corresponding event notification.

Shared storage implementation An application that wants to access a remote storage space declares its interest by using a function whose prototype can be *e.g.* *haggle_create_remote_storage*, without regard to the topology of the underlying network structure. The list of functions that an application can use includes:

- A function to save a file. The application only has to specify the file path or provide a file descriptor. The remaining work is done by the resource sharing framework. There is also an optional parameter where it is possible to specify the preferred security level for the shared file.
- A function to retrieve a previous saved file. This function must return an error code if the file is not currently available. The function must be asynchronous because the lookup, which is performed by sending an *interest* message to the target node holding the data, may require time. Since each file is represented by a single Huggle data object, unique identifiers are used to streamline data retrieval.
- A synchronous function to retrieve a file. This function is used when the event/callback mechanism is not required.
- A function to browse the current reachable shared files.
- A function that allows to modify the security level on saved files. This function can be used when a file is returned to an application, which wishes to store it remotely again after relaxing the security policy (for the sake of fairness).
- A mechanism (*e.g.* callbacks) to store key data for the application in case of scarce resources: that data is then distributed among the members of the community when resource availability rises past pre-set thresholds.
- A function that allows to restore a previous state of the application, or a callback that is invoked when the status is received back.

- A mechanism (*i.e.* callbacks) that allows to automatically store data remotely when the device is out of memory space.

Shared processor implementation Processor sharing functions are included in *libhaggle*.

When an application starts, it advertises its interest in providing a set of common *methods* that can be called from other nodes of the community by registering that set of methods into Haggle. This way, we avoid using a common set of predefined functions and obtain the flexibility of a mobile agent-like mechanism. The drawback is that this mechanism can be used only on nodes running the same application.

The application must assign a priority level to each registered method. This is necessary since each application can contain multiple methods and Haggle requires dispatching them to the remote nodes. In order to keep the application work-flow unaffected, methods with the highest priority (*e.g.* most critical) should be run first.

Shared interface implementation The management and implementation of the interface sharing swap mechanism requires searching for neighbours that can share their network interfaces to reach another node of the community. Consequently, they involve the knowledge and the supervision of the other system resources, namely the Haggle kernel with the collaboration of the resource manager and the connectivity manager. Referring to section 3.1.3.2, the role of the resource manager is to provide information about the level of system resources (*i.e.* battery level), while the role of the connectivity manager is to discover a suitable delegate neighbour, as well as the management and the configuration of the network interface cards selected for the communication.

When it is not possible to switch communication interfaces because a neighbour is not available or because the node has only one network interface, the node should rely on the processor and storage sharing mechanisms in order to conserve energy and/or eventually save the current state.

Global rank computation Referring to section 3.1.3.2, the linear function used for computing the global rank of a node within the community can be expressed as:

$$Globalrank = \sum_{i=1}^N (K_i * (shared/total_avail)) + (T * num_times_shared) \quad (1)$$

where K_i is a constant that can be tuned locally to reflect the importance of a resource for the device, N is the number of shared resources, and T is a constant.

The global rank value is updated every time a new resource sharing process is completed.

In case of concurrent resource sharing requests, the priority of a node to get a resource is given by:

$$Priority = a * Globalrank \quad (2)$$

where a is a constant whose value is comprised between 0 and 1.

Workflow In the previous sections we specified the features provided by the framework for every single resource to be shared. In this section we describe the interactions between managers, applications, and the Hagggle kernel. In a typical working scenario the following steps are performed:

1. Node starts running Hagggle.
2. Hagggle boots.
3. Hagggle managers register for the events that are of interests to them.
4. The resource manager starts monitoring resources of the local node.
5. The resource manager fills up the resource profile of the node. The resource profile is described through metadata (*resource_name-level* pairs) and is added to the node description. Here follows an example:

```
<Resources>
  <item name="processor" value="float_value"/>
  < item name="physical_mem" value="integer_value"/>
  < item name="storage" value="integer_value"/>
  < item name="global_rank" value="float <0.0 to 1.0>"/>
<\Resources>
```

6. The resource manager needs to maintain information about the resources of the node and the nodes that comprise the local community. For this reason, it queries the data manager and node manager, and initializes the correspondent data structures (e.g. the resource/community graph). The node manager keeps track of the list of encountered node(s) and their contact durations.
7. The Hagggle instance on the local node is ready for resource sharing.

The following is the sequence of events that take place when a node n_0 encounters a node n_1 :

- Beacon exchange.
- Node description exchange.
- The node manager of n_0 updates the total contact duration counter of n_1 . When the total contact duration count exceeds a certain threshold (a design parameter), n_1 becomes a potential candidate for the local community.
- The data store is queried and data objects that match interests are exchanged. If n_0 and n_1 share at least one interest and n_1 is a potential candidate, then n_0 adds n_1 to its local community and updates the correspondent data structure.

Newly encountered neighbours and the discovering process play a fundamental role in the creation of the community and in in community data storage and retrieval. The following steps are also performed:

- The resource manager of n_0 checks n_1 's resource profile and checks whether n_1 is a good delegate for resource sharing. Depending on the condition that is satisfied (storage/processor/interface sharing), it sends the correspondent handshake packet.
- If n_1 is a good candidate for storage sharing, then the resource manager of n_0 creates a data object representing a storage sharing request, (be used for the initial handshake) and sends it to n_1 . A similar procedure is employed for processor sharing, in which case, the application manager of n_0 creates a data object that contains the list of the applications along with the list of methods to be run remotely.

The main responsibility of the resource manager is to keep track of the resource level, predict possible scenarios, and try to avoid critical conditions by leveraging on the nodes in the community. A critical threshold value is preset for each resource. Depending on the current level of a given resource, the resource manager can trigger the following actions:

- Save a copy of the current application status. Each application is required by *libhaggle* to provide Haggle with status information to be stored in a remote file system.
- Forwarding computationally expensive operations to members of the community, if the result of the migrating cost analysis suggests taking this action.
- Use the storage space of a community member, if the node is out of local storage space.
- Retrieve an object from remote storage, if local storage becomes available.
- Use the interface sharing swap mechanism.

The data manager is responsible for all the actions that require local storage management. It also exposes interfaces for the management of security policies related to remotely stored data.

3.2 Social-aware content sharing

In this section, we present the final work related to designing, implementing and integrating a social-and context-aware content sharing service in the Haggle architecture. The goal is to demonstrate the advantages of using both context and social information about the user to make the service able to support dynamic reconfiguration of the network and conditions of intermittent connectivity. Thus, defining a novel application of the mobile p2p computing to opportunistic networks. This application demonstrates some of the key features related to the autonomic behaviour of the Haggle nodes, which can self-learn the social context of their users, and self-adapt so as to optimize the resources usage based on this knowledge.

Opportunistic networks represent the emerging pervasive communication paradigm that supports intermittent connectivity scenarios. To develop reliable data exchange and communication, protocols and services can be enhanced by exploiting context and content information, defining a novel data-centric communication paradigm and highly improving the interaction of all layers of the network architecture. This information describes both data that are exchanged through the network (specified by the developed application), and the users carrying

mobile devices, defining their habits, interests and specific information that can be useful in optimizing network services. In this way, we are moving from the standard concept of computer networks to the definition of *networks of people*. In this direction a novel definition of p2p over opportunistic networks has been proposed recently [5], highlighting the importance of disseminating content and context information through the network to improve the autonomous behaviour of mobile devices and to better satisfy the requirements of mobile users.

Substantial work has been done in this area within Hagggle. Specifically, in previous Hagggle deliverables (produced within WP1) a data-centric architecture allowing the design and implementation of application-driven message forwarding in opportunistic networks, reflecting natural ways of interaction between humans, is defined. By exploiting these characteristics, an opportunistic context- and social-aware content sharing service integrated in the Hagggle architecture is presented in which context and content information allows mobile users to share contents even if they will never be directly connected. In the following section the complete integration work related to this activity is described. This complements the description provided in previous architectural deliverables (D1.2 and D1.3), which pertained to the design and the algorithmic aspects of the architecture.

The main idea is to define a general context-aware data dissemination protocol integrated in the Hagggle's architecture and to outline efficient policies of context management useful for the optimization and interaction of Hagggle's resource management protocols. However, to demonstrate the feasibility and the achievable advantages, a content sharing service at the application layer has been developed which considers files as contents to be shared and context information as a combination of user's interests, habits and social contacts. This service is developed on top of the Hagggle architecture which runs on HTC smartphones. It exploits the application API (libhagggleCS) that allows the interaction of the application layer with the main functionalities of Hagggle (e.g., transport protocols, connectivity, forwarding) by simply managing data objects. A demonstration of this work has been done in [6]. To better understand the main features and advantages of this service and context management in general, a detailed description and an application scenario are presented in the following section.

3.2.1 Context definition and cooperative downloads in opportunistic networks

In order to extend the user-generated content model in pervasive environments, this service highlights how using context information enhances the basic features of standard content-sharing applications in opportunistic networks, namely exploiting the nodes' mobility and social contacts to evolve the standard p2p connections among content producers and consumers. Focusing on this specific service, our definition of context mainly refers to the user's shared interests in other users (e.g., genre and type of files they want to share), habits, social contacts and mobility patterns. Since users are mobile and carry devices, and their social behaviour is a valid signal of their mobility patterns, this is potentially useful in predicting future contacts among other users, and improving forwarding decisions. These assumptions are based on social network models [19] and small-world theories [20] where users are grouped in communities, and nodes of the same community have strong social links between each other. Some nodes have also social links outside their "home" community, modelling social relationships with users of different groups. Small-world theories have shown that these "external" links act as shortcuts and enable communications across the network with a small number of hops

(six hops in the "classical" small-world models). Our service exploits this behavioural model to spread context information across the users' communities allowing network protocols and applications to improve their performances by correctly evaluating the users' behaviour.

Two different notions of context related to a single user are introduced in the application presented. The first is the *private context* of a user, which consists of personal information¹ (e.g., name, address, job, communities he/she belongs to) that is not related to the developed service (and thus can be used as general context to be shared among several local applications). The second is the *public context*, which is mainly defined by the service (e.g., sharing interests in terms of files genre, category, content attributes). All this information is collected in an object called *Identity Table* (IT), since it identifies both characteristics of the single user, and his/her interactions with the service. The user initializes it directly through a user-friendly GUI on the mobile device. Even though all this information can be represented by attribute values (name/value pairs), the IT can contain a large number of attributes that can overload Haggle's metadata. This requires a significant filtering effort to match all possible managers' interests in the single context attributes. Thus, the IT is spread on the network through Haggle as a single data object dedicated to context information, and a specific module in the architecture is in charge of managing this information.

Following the model presented in [3] for a context-aware forwarding algorithm in opportunistic networks, ITs are exchanged among 1-hop neighbours. Therefore, by exchanging ITs during the neighbour discovery procedure, each node knows the personal and public information of all its neighbours, adds them to its *current context*, and obtains thus a snapshot of the surroundings. Since users usually move from one community to another, it is also important to locally maintain the *context evolution* over time (i.e., information about context of past encountered users as the evolution of social contacts of the local user). Considering users move from one community to another, generally returning to a "home community", the information related to past neighbours and related contexts can be used in a different context to select contents that can be useful or interesting for users belonging to a previous context. This is implemented in content sharing as an autonomous search and storage by the local node for contents in the current context that match the interests of past encountered users, assuming that they will re-encounter each other in the future. In this way, nodes use part of their local resources to pre-fetch data to satisfy possible future requests of past neighbours. This procedure has been called *cooperative downloading* among opportunistic nodes.

The possible choices between the files a user can download in favour of previous neighbours depends mostly on the local resource available and also the data utility. Intuitively, the main idea is to select those files that maximize the utility for both the local user (e.g. data belonging to the same category she is interested in) and the users belonging to the different communities he/she also belongs to. We can thus define the following utility function:

$$U = u(l) + \sum_i \omega_i * u_i(c) \quad (3)$$

where $u(l)$ is the utility for the local user, $u_i(c)$ is the utility for the i -th community the user is in contact with, and ω_i is a cooperation index that defines the willingness of the user to cooperate with the i -th community (i.e., to spend own resources to increase data availability for that community). Note that, by using cooperation indexes greater than 0, one can avoid the

¹Managing private/personal information about users requires dealing with privacy issues

selfish users behaviour in which they tend to maximize only their own utility. In addition, the autonomous cooperative downloads do not involve the files the local user is interested in, since he/she can directly request them. Another component of the utility computation is represented by the current context of the local user. Since it is highly probable that users visiting the same community are in communication range, or can directly download the files they are interested in with few-hops connections, we can assign a lighter weight to the current context of the local user, so that the system can privilege the other contexts.

After this preliminary analysis, it is necessary to define the parameters the utility function depends on. Generally, we consider the *access probability* as the probability the local node has to successfully download the specific file, which is strictly related to the *context stability* as a function of the time the local node would spend in the current context, and the mobility model of its neighbours. Furthermore, we must consider the benefits the users of the examined context can obtain from the cooperative downloading by the local user. Finally, we must consider the availability of local resources for each possible download. A detailed analysis of the *context-aware utility* and related parameters can be found in [4]. In our content-sharing service we implemented a simplified form of this function to highlight the main advantages of using context, the reliability of the application in case of intermittent connectivity, and the reduction of dissemination requests over the network. The application scenario and a practical example are shown in the following section.

3.2.2 Application scenario

Assuming that users belonging to the same community share some common interests and have social relationships outside their main community occasionally "visiting" other communities, we can present the following example. Two social communities of users share interests in specific file categories as shown in Figure 6(a). In the example, user C belongs to both communities, and users share the following interests in both: on the left side users A and C are interested in mp3 and jpeg files, while user B is interested in jpeg and avi files; on the right side both users D and E are interested in jpeg and avi. Assuming that all nodes in each community are able to communicate with each other, after the neighbor discovery phase each of them knows the context information of the others (IT) *i.e.*, the interests and the list of shared files. This initializes the current context. As user C belongs to both communities, it is highly probable that he/she will move towards the right-hand-side community and can help distributing files to interested users. When he/she reaches the right-hand-side community, he/she is able to select the of interest in and download them on his/her device. Moreover, looking at the previously stored context information, the application run by user C is also able to identify, and possibly download, additional files which the users of the left-hand-side community are probably interested in (see Figure 6(b)), therefore implementing the cooperative download procedure. In this example, simple context information related to the shared contents is used to improve the efficiency of this service in opportunistic environments.

In order to implement in Huggle the application scenario described above, we need a general framework to provide context information to managers and modules, and to utilize context information in order to select best destinations for a specific content. An overview of this general framework is given in the following section.

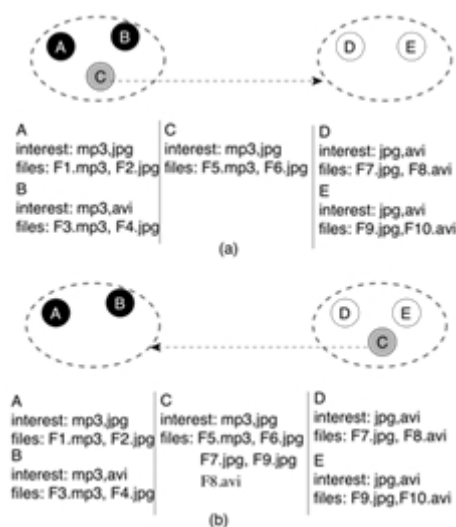


Figure 6: Example of cooperative download

3.2.3 Integration within the Haggle architecture

The content sharing application, as well as the data dissemination mechanisms described above, have been integrated within the Haggle architecture. This has been achieved partly by defining a Data Dissemination (DD) module belonging to the resource manager of the reference architecture (see Figure 7), and by implementing specific application logic by exploiting the libhaggle API. Context awareness has been integrated as follows. As previously defined, we mainly refer to two different notions of context: private context and public context. Private context is defined by general and personal information about the user carrying the mobile device and by physical characteristics of the device itself, and to improve the autonomous behaviour of the system according to the user behaviour and the surrounding environment. Public context is defined by a specific application to improve its own purposes. From an architectural standpoint, this differentiation allows for full flexibility, as it is possible to share context information with other applications, as well as to keep some context information private to a specific application. On the one hand, the management of the public context information is the responsibility of each single application, which declares its interests in the data object containing this information, and directly process it for specific application purposes. For example, the content sharing application defines as private context the list of shared files, file genres the user is interested in, and specific attributes the user associates to a file. On the other hand, the Data Dissemination module (DD) is designed to manage the private context (Figure 7). This module is in charge of implementing the context-aware data dissemination protocol that exploits context information in order to select best destinations for a specific content. It thus maintains a strict interaction with the related application, and, at the same time, it provides context information to other managers and modules inside Haggle depending on their specific interests. Thanks to this module, it is possible to share context information among different applications and modules, when appropriate and needed.

Since most part of the context information is strictly related to the user and his/her interactions with one or more applications, a user-friendly graphical interface is also necessary to initialize



Figure 7: Context-aware data dissemination module



Figure 8: Screenshots of the social-aware content sharing service GUI

this information. An example is shown in Figure reffig:context-aware-GUI. The information selected and inserted by the user is passed to the Data Dissemination module, which stores it in the data store in the form of an Identity Table (IT) and spreads it across the network during the neighbour discovery phase. When a data object containing the IT is received by a Huggle node, the Data Dissemination module examines it, stores the information related to the public context and forwards the private context to the interested applications.

3.3 Security

In the ADULT-Huggle architecture, new security features are defined. In addition to the existing secure forwarding solutions fulfilling the privacy model 2 (see deliverable D4.2), in the ADULT-Huggle specification nodes that do not share any common attribute with the source and the destination nodes can discover matching encrypted attributes and can correctly forward the message to the next hops. Indeed, the new version of *Huggle certificates* allows the encryption of the {attribute_name, attribute_value} pair, which can only be discovered by nodes

with the same attributes. Thanks to these new Haggle certificates, all nodes, no matter their community membership, are now able to compare certificates and discover if some attributes match in order to correctly forward packets. Moreover, the setup phase, during which a node retrieves its attribute certificate from the issuer, is also defined.

3.3.1 Encrypted Haggle Certificates

Haggle certificates, as defined in the YOUNG-Haggle architecture, are useful to prove the authenticity of a given attribute name/value tuple, and therefore enable nodes to take correct forwarding decisions based on these attributes: when a node presents a valid certificate and if the issuer of this certificate is trustworthy, it means that the node has the attribute name/value it claims, and that it did not forge it. Nonetheless, the attributes name/value can sometimes contain sensitive information: in context-based forwarding like PROPICMAN or HiBoP ([9, 3]) the attribute names and values refer to the node's owner location (address, workplace,...) or other personal information like age or hobbies. Therefore, it is important to keep this information private, and to manage the set of nodes that are authorized to view them. In deliverable D4.2, we defined three privacy models:

- Model 1: privacy oblivious. In this model, nodes do not care about the privacy of their information and therefore, disclose their attributes to other nodes.
- Model 2: intra-community privacy. In this model, nodes reveal all their attributes only to nodes in their community.
- Model 3: full privacy. In this model, nodes do not reveal an attribute to any other node, unless the nodes share the same attribute.

In the YOUNG-Haggle architecture, a solution ensuring secure forwarding pertaining to privacy model 2 was proposed: the ADULT-Haggle security manager features a light solution for privacy model 3. This solution requires the attribute name and value of the Haggle certificate to be encrypted. In this regard, the issuer of the certificate encrypts the attribute name and value using a standard symmetric key algorithm. Nodes can then securely exchange their certificates and only nodes sharing the same attributes discover that they share the same attribute. Moreover, in privacy model 3, nodes that do not share the same attributes are still able to compare encrypted certificates and hence forward encrypted data objects to interested nodes without discovering any additional information.

3.3.2 Definition of the setup phase

Like classical certificates, attribute certificates need first to be generated and certified by an issuer, also referred to as trusted party. The level of trust that a node gives to a certificate directly depends on the trust it has in the certificate issuer. In the YOUNG-Haggle architecture, attribute certificates were predefined and already stored in memory. In the ADULT-Haggle, the communication protocol between the issuer and any requesting Haggle node is implemented. This setup phase is particularly interesting for community-based communication protocols where an initiator node I decides to initiate a community locally: I provides certificates signed with its

private key to other nodes that it trusts. The complete protocol includes all message exchanges to obtain the certificate, including challenge request and response, and then the signature of the certificate. More details can be found in deliverable D4.3.

3.3.3 Securing the communication channel

In Huggle, since messages may be broadcast (in the form of data objects), an eavesdropper can listen to the channel and get a copy of the message which can leak valuable information. Indeed, the encryption of attribute certificates is not enough to avoid a malicious node to retrieve some useful information: even when attribute certificates are encrypted, the result of the encryption of an attribute is always the same; therefore, a malicious node can discover whether some nodes share certain attributes or not, without discovering the content of the attribute. In order to avoid this problem, the communication channel between neighbours must be secured and the entire data object needs to be encrypted. Therefore, before a message is sent, a node encrypts the complete data object with a symmetric key, which in turn is encrypted with the public key of the destination node and is then sent along with the encrypted data object.

More details on the various encryption mechanisms and their implementation are provided in deliverable D4.3.

4 Conclusions

In this document we have presented the ADULT-Haggle specification, which consists of several enhancements to the YOUNG-Haggle architecture along with the integration of certain mechanisms already described in the past architectural deliverables. We started from an experimental evaluation of the YOUNG-Haggle architecture and moved on to focus on the improvements to resource management and security.

The final Haggle architecture is able to support opportunistic communication among nodes within a community, to act and react with the support of its own data, to exploit context information for data forwarding and dissemination, and to secure a data exchange among nodes.

References

- [1] <http://www.rentacoder.com>.
- [2] [http://msdn.microsoft.com/en-us/library/kwdt6w2k\(VS.71\).aspx](http://msdn.microsoft.com/en-us/library/kwdt6w2k(VS.71).aspx).
- [3] C. Boldrini, M. Conti, and A. Passarella. Exploiting users' social relations to forward data in opportunistic networks: The hibop solution. In *Pervasive and Mobile Computing*, June 2008.
- [4] M. Conti, C. Boldrini, and A. Passarella. Contentplace: social-aware data dissemination in opportunistic networks. 2008.
- [5] M. Conti, F. Delmastro, and A. Passarella. Context-aware p2p over opportunistic networks. In *Mobile Peer-to-Peer Computing for Next Generation Distributed Environments: Advancing Conceptual and Algorithmic Applications*, 2009.
- [6] M. Conti, F. Delmastro, and A. Passarella. Social-aware content sharing in opportunistic networks. In *IEEE SECON 2009*, 2009.
- [7] L. Danon, J. Duch, and et al. Comparing community structure identification. *J. Stat. Mech*, page P09008, October 2005.
- [8] T. N. Ellahi, B. Hudzia, L. McDermott, and T. Kechadi. Transparent migration of multi-threaded applications on a java based grid. *AND SERVICES*, 2006.
- [9] A. Puiatti H. A. Nguyen, S. Giordano. Probabilistic routing protocol for intermittently connected mobile ad hoc networks (propicman). Helsinki, Finland, June 2007. IEEE AOC.
- [10] P. Hui, J. Crowcroft, and E. Yoneki. Bubble rap: social-based forwarding in delay tolerant networks. In *Proceedings of the 9th ACM international symposium on Mobile ad hoc networking and computing (MobiHoc)*, 2008.
- [11] P. Hui, E. Yoneki, S. Chan, and J. Crowcroft. Distributed community detection in delay tolerant networks. In *MobiArch'07*, 2007.
- [12] Paulo Marques, Paulo Simoes, Luis Silva, Fernando Boavida, and Joao Silva. Providing applications with mobile agent technology. In *in Proc. 4th IEEE International Conference on Open Architectures and Network Programming (OpenArch'01)*, pages 129–136. Press, 2001.
- [13] M. Newman. Detecting community structure in networks. *Eur. Phys. J.B.*, (38):321–330, 2004.
- [14] C. J. Patten, K. A. Hawick, and J F. Hercus. Towards a scalable metacomputing storage service. In *In Proceedings of the 7th International Conference on High Performance Computing and Networking Europe (HPCN99)*, 1999.
- [15] Ichiro Satoh. The design and implementation of the mobilespheres mobile agent system, 2000.
- [16] T. Schoberth, J. Preece, and A. Heinzl. Online communities: A longitudinal analysis of communication activities. In *Proceeding of HICSS36*, 2003.

- [17] Kazuyuki Shudo and Yoichi Muraoka. Noncooperative migration of execution context in java virtual machines. In *In Proc. of the First Annual Workshop on Java for High-Performance Computing (in conjunction with ACM ICS'99)*, 1999.
- [18] P. Sridharan. *Java Network Programming*. Prentice-Hall, 1997.
- [19] S. Wasserman and K. Faust. *Social network analysis: Methods and applications*. Cambridge University Press, 1994.
- [20] D.J. Watts. *Small worlds: The dynamics of networks between order and randomness*. Princeton University Press, 1999.