INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Team LogNet

Logical Networks: Self-organizing Overlay Networks and Generic Overlay Computing Systems

Sophia Antipolis - Méditerranée

THEME COM

Activity Report

2008

Table of contents

1. Team **1**
2. Overall Objectives **1**
2.1. LogNet's Motto and Logo 1
2.2. Overall objectives 2
2.3. Highlights 2
3. Scientific Foundations **3**
3.1. Lognet's general context 3
3.2. General definitions 4
3.3. Background 1: Arigatoni overlay network computer 5
3.3.1. Arigatoni units 5
3.3.2. Virtual organizations 7
3.3.3. Resource discovery protocol (RDP) 7
3.3.4. Virtual Intermittent Protocol (VIP) 8
3.3.5. Two simple examples 8
3.4. Background 2: SmartTools service-oriented overlay software factory 10
3.4.1. SmartTools units 10
3.4.2. SmartTools into Eclipse 10
3.4.3. The SmartTools' evolution 11
3.4.4. The new SmartTools' functional units 11
3.5. General research directions 11
3.5.1. On Virtual organizations 13
3.5.2. On Resource discovery 13
3.5.3. Execution model 14
4. Application Domains **16**
4.1. Panorama 16
4.2. Potential applications 17
5. Software **18**
5.1. Ariwheels 18
5.1.1. Simulator 18
5.1.2. Network client 19
5.2. PiNet 21
5.3. ArchiNet 22
5.4. SmartTools 23
5.5. Arigatoni simulator 23
5.6. BabelChord simulator 23
5.7. iRho and Snake interpreters 24
6. New Results **25**
6.1. Programmable overlay networks with ArchiNet 25
6.2. Strong typed communication with PiNet 25
6.3. DHT formal specification 26
6.4. Social overlay networks 26
6.4.1. Features 27
6.4.2. Suitable applications 27
6.5. P2P resource discovery 27
6.6. SmartTools into Eclipse 28
6.7. Ariwheels: an overlay architecture for vehicular networks 28
6.8. Powerful resource discovery for Arigatoni 30
6.9. Conditional Logical Framework 30
6.10. Adding trait inheritance to the Java language 31

6.11. iRho: An Imperative Rewriting-calculus 32
7. Other Grants and Activities **34**
7.1. Regional Initiatives 34
7.2. National Initiatives 34
7.3. European Initiatives 34
7.3.1. Ph.D. Exchanges founded by Italian Research Founding Agency 34
7.3.2. FP6 FET Global Computing: IST AEOLUS, 2005-2009 34
7.3.3. FP6 TEMPUS, 2007-2009 34
7.4. Visitors 35
7.4.1. IN 35
7.4.2. OUT 35
8. Dissemination **35**
8.1. Participation in committees 35
8.2. Workshop organization 35
8.3. Participation in conference committees 36
8.4. Theses 36
8.5. Referees 36
8.6. Teaching 36
8.7. Invited talks 36
8.8. Participation in scientific meetings 36
8.9. Participation in conferences 37
8.10. Spare presentations 37
9. Bibliography **37**

LogNet is an Inria team. The team has been created on January the 1st, 2008.

1. Team

Research Scientist

Luigi Liquori [Head, Research associate, CR INRIA, HdR]
Didier Parigot [Research associate, CR INRIA, HdR]
Bernard Serpette [Research associate, CR INRIA]
Michel Cosnard [CEO INRIA, external collaborator]

Technical Staff

Baptiste Boussemart [INRIA engineer (until September 2009)]

PhD Student

Francesco Bongiovanni [INRIA-PACA fellow (since October 1st 2008), defense planned in 2011]
Petar Maksimovic [INRIA-TEMPUS fellow (since December 1st 2008), defense planned in 2011]

Post-Doctoral Fellow

Cedric Tedeschi [INRIA postdoctoral fellown (since October 1st 2008)]

Administrative Assistant

Corinne Mangin [INRIA]

2. Overall Objectives

2.1. LogNet's Motto and Logo

Our Motto is *"Computer is moving on the edge of the Network..."* by Jan Bosch, Nokia Labs, [LNCS 4415, 2007] and our logo is in Figure 1.

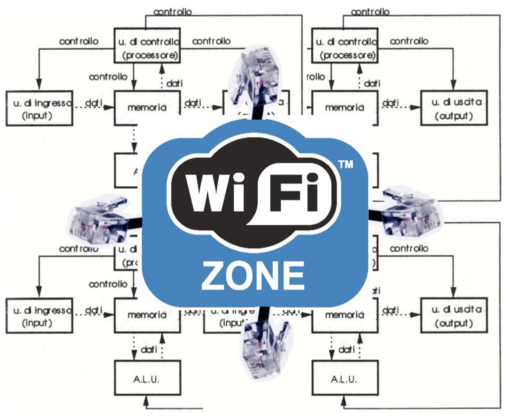
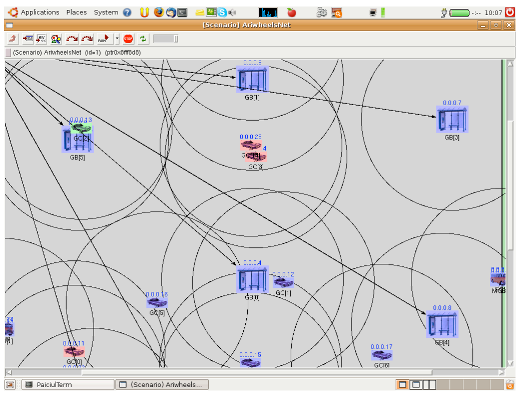


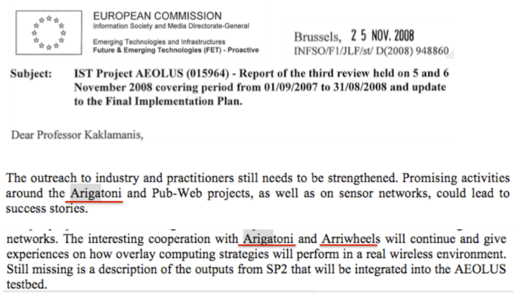
Figure 1. Our logo

2.2. Overall objectives

We propose foundations for *generic overlay networks* and *overlay computing systems*. Such overlays are built over a large number of distributed *computational agents*, virtually organized in *colonies*, and ruled by a leader (*broker*) who is elected democratically (*vox populi, vox dei*) or imposed by system administrators (*primus inter pares*). Every agent asks the broker to log in the colony by declaring the resources that can be offered (with variable guarantees). Once logged in, an agent can ask the broker for other resources. Colonies can recursively be considered as *evolved agents* who can log in an outermost colony governed by another super-leader. Communications and routing intra-colonies goes through a broker-2-broker *PKI*-based negotiation. Every broker routes intra- and inter- *service requests* by filtering its *resource routing table*, and then forwarding the request first inside its colony, and second outside, via the proper super-leader (thus applying an *endogenous-first-estrogen-last* strategy). Theoretically, queries are formulæ in first-order logic equipped with a small program used to *orchestrate* and *synchronize* atomic formulæ (atomic services). When the client agent receives notification of all (or part of) the requested resources, then the real resource exchange is performed directly by the server(s) agents, without any further mediation of the broker, in a pure peer-to-peer fashion. The proposed overlay promotes an *intermittent* participation in the colony, since peers can appear, disappear, and organize themselves dynamically. This implies that the routing process may lead to *failures*, because some agents have quit or are temporarily unavailable, or they were logged out *manu militari* by the broker due to their poor performance or greediness. We aim to design, validate through simulation, and implement these foundations in a generic overlay network computer system.

2.3. Highlights





EUROPEAN COMMISSION
Information Society and Media Directorate-General

Emerging Technologies and Infrastructures
Future & Emerging Technologies (FET) - Proactive

Brussels, 25 NOV. 2008
INFSO/F1/JLF/st/ D(2008) 948860

Subject: **IST Project AEOLUS (015964) - Report of the third review held on 5 and 6 November 2008 covering period from 01/09/2007 to 31/08/2008 and update to the Final Implementation Plan.**

Dear Professor Kaklamanis,

The outreach to industry and practitioners still needs to be strengthened. Promising activities around the Arigatoni and Pub-Web projects, as well as on sensor networks, could lead to success stories.

networks. The interesting cooperation with Arigatoni and Arriwheels will continue and give experiences on how overlay computing strategies will perform in a real wireless environment. Still missing is a description of the outputs from SP2 that will be integrated into the AEOLUS testbed.

Figure 2. The Ariwheels simulator and the European Commission point of view.

- The *Ariwheels* overlay network is being proposed as a publish & subscribe protocol in the vehicular platform under development in the VICSUM project (2Meur, founded by the Regione Piemonte) led by Politecnico di Torino and involving the Centro Ricerche Fiat (CRF) and the Centro Supercalcolo Piemonte [20], [22]. (Successful) results will be suitable to be integrated in the new *BLUETOOTH* system device *Blue&Me*^{TM} by Fiat&Microsoft. The project will exploit the availability of existing urban infrastructure and public transportation vehicles (pledged by *GTT* – Gruppo Torinese Trasporti, the Torino's public bus and metro In the simulator the color's semantics is:
  - BLUE = Car already logged to a Broker (Bus Stop).
  - PINK = Car that has lost *WIFI* connection with the Broker (Bus Stop).
  - GREEN = Car that is looking for a Broker (Bus Stop).

- This year, the *Arigatoni* and the *Ariwheels* projects have been highlighted in the third year report of the IST Project AEOLUS covering period from 01/09/2007 to 31/08/2008 (Figure 2).

3. Scientific Foundations

3.1. Lognet's general context

Keywords: *Overlay networks*, *query routing*, *resource discovery*, *social networks*, *virtual organizations*.

Participants: Luigi Liquori, Didier Parigot, Bernard Serpette.

The explosive growth of the Internet gives rise to the possibility of designing large *overlay networks* and *virtual organizations* consisting of Internet-connected computers units, able to provide a rich functionality of services that makes use of aggregated computational power, storage, information resources, etc. We would like to start our first activity report with the standard definition of *Computer System*.

Definition 1 (Computer System)
A computer system is composed by a computer hardware and a computer software.

- A Computer Hardware is the physical part of a computer, including the digital circuitry, as distinguished from the computer software that executes within the hardware. The hardware of a computer is infrequently changed, in comparison with software and data.
- A Computer Software is composed by three parts, namely, system software, program software, and application software.
  - The System Software helps run the computer hardware and computer system. Examples are operating systems (OS), device drivers, diagnostic tools, servers, windowing systems...
  - The Program Software usually provides tools to assist a programmer in writing computer programs and software using different programming languages. Examples are text editors, compilers, interpreters, linkers, debuggers for general purpose languages...
  - The Application Software allows end users to accomplish one or more specific (non computer related) tasks industrial automation, business software, educational software, medical software, databases, computer games...

Starting from the previous basic skeleton definition, we elaborate the LogNet's vision of what an *Overlay Network Computer System* is. The reader can focus on the tiny but crucial differences.

Definition 2 (Overlay Computer System)
An overlay computer system is composed by an overlay computer hardware and an overlay computer software.

- An Overlay Computer Hardware is the physical part of an overlay computer, including the digital circuitry, as distinguished from the overlay computer software that executes within the hardware. The hardware of an overlay computer changes frequently and it is distributed in space and in time. Hardware is organized in a network of collaborative computing agents connected via *IP* or ad-hoc networks; hardware must be negotiated before being used.
- An Overlay Computer Software is composed by three parts, namely, overlay system software, overlay program software, and overlay application software.
  - The Overlay System Software helps run the overlay computer hardware and overlay computer system. Examples are network middleware playing as a distributed operating system (dOS), resource discovery protocols, virtual intermittent protocols, security protocols, reputation protocols...

- The Overlay Program Software usually provides tools to assist a programmer in writing overlay computer programs and software using different overlay programming languages. Examples are compilers, interpreters, linkers, debuggers for workflow-, coordination-, and query-languages.
- The Overlay Application Software allows end users to accomplish one or more specific (non-computer related) tasks industrial automation, business software, educational software, medical software, databases, and computer games...Those classes of applications deal with computational power (*Grid*), file and storage retrieval (*P2P*), web services (*Web2.0*), band-services (*VoIP*), computation migrations...

Therefore, LogNet's objectives can be summarized as follows:

- to provide adequate notions and definitions of a generic overlay network computer; from a desktop distributed calculator to a programmable distributed overlay computer;
- on the basis of these definitions, to propose a precise architecture of a generic overlay network computer and implement it;
- on the basis of these definitions, to implement an overlay software factory suitable to help the logical and software assembling of an overlay network computer.

3.2. General definitions

An overlay network is a computer network which is built on top of another network. Overlay networks can be constructed in order to permit routing messages to destinations not specified by an *IP* address. In what follows, we briefly describe the main entities underneath a virtual organization.

Agents. An agent in the overlay is the basic computational entity of the overlay: it is typically a device, like a *PDA*, a laptop, a *PC*, or smaller devices, connected through *IP* or other *ad hoc* communication protocols in different fashions (wired, wireless). Agents in the overlay can be thought of as being connected by virtual or logical links, each of which corresponds to a path, through many physical links, in the underlying network. For example, many peer-to-peer networks are overlay networks because they run on top of the Internet.

Colonies and colony's leaders. Agents in the overlay are regrouped in *Colonies*. A colony is a simple virtual organization composed by exactly one *leader*, offering some broker-like services, and some set of *agents*. The leader, being also an agent, can be an agent of a colony different of the one it manages. Thus, agents are simple computers (think it as an *amoeba*), or sub-colonies (think it as a *protozoa*). Every colony has *exactly* one leader and at least one agent (the leader itself). Logically an agent can be seen as a *collapsed colony*, or a *leader managing itself*. The leader is the only one who knows all agents of its colony. One of the tasks of the leader is to manage (un)subscriptions to its colony.

Resource discovery. By adhering a colony, an agent can expose resources it has and/or ask for resources it needs. Another task of a leader is to manage the resources available in its colony. Thus, when an agent of the overlay needs a specific resource, it makes a request to its leader. A leader is devoted to contacting and negotiating with potential servers, to authenticating clients and servers, and to route requests. The rationale ensuring scalability is that every request is handled first inside its colony, and then forwarded through the proper super-leader (thus applying an *endogenous-first-exogenous-last* strategy).

Orchestration. When an agent receives an acknowledgment of a service request from the direct leader, then the agent is served directly by the server(s) agents, *i.e.* without a further mediation of the leader, in a pure *P2P* fashion. Thus, the "main" program will be run on the agent computer machine that launched the service request and received the resources availability: it will orchestrate and coordinate data and program resources executed on others agent computers.

3.3. Background 1: Arigatoni overlay network computer

As suggested by our previous definitions, we are mainly concerned by three topics: network organization, resource discovery and orchestration. These topics are studied in complementary way by *Arigatoni* (work started by Luigi Liquori and Michel Cosnard) and *SmartTools* and *ArchiNet* and *PiNet* (work started by Didier Parigot and Bernard Serpette). Indeed *Arigatoni* and *SmartTools* are both built around a similar concept of virtual organization (colonies of agents for *Arigatoni* and colonies of "software components" for *SmartTools*). In this section and the next one we will describe the current status of *Arigatoni* and *SmartTools*.

The *Arigatoni* overlay network computer [1], [2], [10], [9], [4], [5], [11], and [14] developed since 2006 in the Mascotte Project Team by Luigi Liquori and Michel Cosnard, and then in the LogNet team, is a structured multi-layer overlay network which provides resource discovery with variable guarantees in a virtual organization where peers can appear, disappear, and self-organize themselves dynamically. *Arigatoni* is *universal* in the sense of Turing machines, or *generic* as the von Neumann computer architecture is.

Every agent asks the broker to log in the colony by declaring the resources that it provides (with variable guarantees). Once logged in, an agent can ask the broker for other resources. Colonies can recursively be considered as *evolved agents* who can log in an outermost colony governed by another super-leader. Communications and routing intra-colonies go through a broker-2-broker *PKI*-based negotiation. Every broker routes intra- and inter- *service requests* by filtering its *resource routing table*, and then forwarding the request first inside its colony, and second outside, via the proper super-leader (thus applying an *endogenous-first-estrogen-last* strategy).

Theoretically, queries are formulæ in first-order logic. When the client agent receives notification of all (or part of) the requested resources, then the real resource exchange is performed directly by the server(s) agents, without any further mediation of the broker, in a pure peer-to-peer fashion. The proposed overlay promotes an *intermittent* participation in the colony. Therefore, the routing process may lead to *failures*, because some agents have quit, or are temporarily unavailable, or they were logged out by the broker due to their poor performance or greediness.

Arigatoni features essentially two protocols: the *resource discovery protocol* dealing with the process of an agent broker to find and negotiate resources to serve an agent request in its own colony, and the *virtual intermittent protocol* dealing with (un)registrations of agents to colonies.

Dealing essentially with resource discovery and peers' churn has one important advantage: the complete generality and independence of any offered and requested resource. *Arigatoni* can fit with various scenarios in the global computing arena, from classical *P2P* applications (file- or bandwidth-sharing), to new *Web2.0* applications, to new *V2V* and *V2I* over *MANET* applications, to more sophisticated *Grid* applications, until possible, futuristic *migration computations*, *i.e.* transfer of a non-completed local run to another agent, the latter being useful in case of catastrophic scenarios, like fire, terrorist attack, earthquake, etc.

Since 2008, Bernard Serpette is working on the foundation and implementations of an overlay network computer based on an evolution of the *Arigatoni* network model (see Software Section and New Results).

3.3.1. Arigatoni units

In what follows, we briefly introduce the logic units underneath a generic overlay network. Peers' participation in *Arigatoni*'s colonies is managed by the *Virtual Intermittent Protocol (VIP)*; the protocol deals with the *dynamic topology* of the overlay, by allowing agent computers to login/logout to/from a colony (using the *SREG* message). Due to this high node churn, the routing process may lead to *failures*, because some agents have logged out, or because they are temporarily unavailable, or because they have logged out *manu militari* by the broker for their poor performance or greediness.

The total decoupling between peers in *space* (peers do not know other peers' locations), *time* (peers do not participate in the interaction at the same time), *synchronization* (peers can issue service requests and do something else, or may be doing something else when being asked for services), and *encapsulation* (peers do not know each other) are key features of *Arigatoni*'s scalability.

Agent computer (AC). This unit can be, *e.g.*, a cheap computer device composed by a small *RAM-ROM-HD* memory capacity, a modest *CPU*, a ≤ 40 keystrokes keyboard (or touchscreen), a tiny screen (≤ 4 inch), an *IP* or *ad hoc* connection (via *DHCP*, *BLUETOOTH*, *WIFI*, *WIMAX*...), a *USB* port, and very few programs installed inside, *e.g.* one simple editor, one or two compilers, a `mail` client, a mini browser... Our favorite device actually is the Internet terminal *Nokia N810*. Of course an *AC* can be a supercomputer, or an high performance *PC*-cluster, a large database server, an high performance visualizer (*e.g.* connected to a virtual reality center), or any particular resource provider, even a *smart-dust*. The operating system (if any) installed in the *AC* is not important. The computer should be able to work in *local mode* for all the tasks that it could do locally, or in *global mode*, by first registering itself to one or many colonies of the overlay, and then by asking and serving global requests via the colony leaders. In a nutshell, the tasks of an *AC* are:

- Discover the address of one or many agent brokers (*AB*s), playing as colony leaders, upon its arrival of itself in a "connected area"; this can be done using the underlay network and related technologies;
- Register on one or many *AB*s, so entering *de facto* the *Arigatoni*'s virtual organization;
- Ask and offer some services to others *AC*s, via the leaders *AB*s;
- Connect directly with other *AC*s in a *P2P* fashion, and offer/receive some services. Note that an *AC* can also be a resource provider. This symmetry is one of the key features of *Arigatoni*. For security reasons, we assume that all *AC* come with their proper *PKI* certificate.

Agent Broker (AB). This unit can be, *e.g.*, a computer device made by a high speed *CPU*, an *IP* or *ad hoc* connection (via *DHCP*, *BLUETOOTH*, *WIFI*, *WIMAX*...), a high speed hard-disk with a *resource routing table* to route queries, and an efficient program to match and filter the routing table. The computer should be able to work in *global mode*, by first registering itself in the overlay and then receiving, filtering and dispatching global requests through the network. The tasks of an *AB* are:

- Discover the address of another *super-AB*, representing the *super-leader* of the *super-colony*, where the *AB* colony is embedded. We assume that every *AB* comes with its proper *PKI* certificate. The policy to accept or refuse the registration of an *AC* with a different *PKI* is left open to the level of security requested by the colony;
- Register/unregister the proper colony on the *leader AB* which manages the super-colony;
- Register/unregister clients and servants *AC* in its colony, and update the internal resource routing table, accordingly;
- Receive the request of service of the client *AC*;
- Discover the resources that satisfy an *AC* request in its local base (local colony), according to its resource routing table;
- Delegate the request to an *AB* leader of the direct super-colony in case the resource cannot be satisfied in its proper colony; it must register itself (and by product its colony) to another super-colony;
- Perform a combination of the above last two actions;
- Deal with all *PKI* intra- and inter-colony policies;
- Notify, after a fixed *timeout period*, or when all *AC*s failed to satisfy the delegated request, to the *AC* client the *denial of service* requested by the *AC* client;
- Send all the information necessary to make the *AC* client able to communicate with the *AC* servants. This notification is encoded using the resource discovery protocol. (Finally, the *AC* client will directly talk with the *AC*s servants).

Agent Router (AR). This unit implements all the low-level overlay network routines, those which really have access to the *IP* or to the *ad-hoc* connections. In a nutshell, an *AR* is a shared library dynamically linked with an *AC* or an *AB*. The *AR* is devoted to the following tasks:

- Upon the initial start-up of an *AC* (resp. *AB*) it helps to register the unit with one or many *AB* that it knows or discovers;
- Checks the well-formedness and forwards packets of the two *Arigatoni*'s protocols across the overlay toward their destinations.

3.3.2. Virtual organizations

Agent computers communicate by first registering to the colony and then by asking and offering services. The leader agent broker analyzes service requests/responses, coming from its own colony or arriving from a surrounding colony, and routes requests/responses to other agents. Agent computers get in touch with each other without any further intervention from the system, in a *P2P* fashion. Peers' coordination is achieved by a simple program written in an orchestration/business language *à la BPEL*, or *JOpera*.

Symmetrically, the leader of a colony can arbitrarily unregister an agent from its colony, *e.g.*, because of its bad performance when dealing with some requests or because of its high number of "embarrassing" requests for the colony. This strategy, reminiscent of the Roman *do ut des*, is nowadays called, in Game Theory, Rapoport's *tit-for-tat* strategy [46] of cooperation based on reciprocity. Tit-for-tat is commonly used in economics, social sciences, and it has been implemented by a computer program as a winning strategy in a chess-play challenge against humans (see also the well known *prisoner dilemma*). In computer science, the tit-for-tat strategy is the stability (*i.e.* balanced uploads and downloads) policy of the *Bittorrent P2P* protocol.

Once an agent computer has issued a request for some service, the system finds some agent computers (or, recursively, some sub-colonies) that can offer the resources needed, and communicates their identities to the (client) agent computer as soon as they are found.

The model also offers some mechanisms to dynamically adapt to *dynamic topology changes* of the overlay network, by allowing an agent (computer or broker, representing a sub-colony) to login/logout in/from a colony. This essentially means that the process of routing request/responses may lead to failure, because some agents logged out or because they are temporarily unavailable (recall that agents are not slaves). This may also lead to temporary denials of service or, more drastically, to the complete logout of an agent from a given colony in the case where the former does not provide enough services to the latter.

3.3.3. Resource discovery protocol (RDP)

Kind of discovery. The are mostly two mechanisms of resource discovery, namely:

- The process of an *AB* to find and negotiate resources to serve an *AC* request in its own colony;
- The process of an *AC* (resp. *AB*) to discover an *AB*, upon physical/logical insertion in a colony.

The first discovery is processed by *Arigatoni*'s resource discovery protocol, while the second is processed out of the *Arigatoni* overlay, using well-known network protocols, like *DHCP*, *DNS*, the service discovery protocol *SLP* of *BLUETOOTH*, or Active/Passive Scanning in *WIFI*.

The current *RDP* protocol version allows the request for *multiple services* and *service conjunctions*. Adding service conjunctions allows an *AC* to offer several services at the same time. Multiple services requests can be also asked to an *AB*; each service is processed sequentially and independently of others. As an example of multiple instances, an *AC* may ask for three *CPU*s, *or* one chunk of *10GB* of *HD*, *or* one gcc compiler. As an example of a service conjunction, an *AC* may ask for another *AC* offering *at the same time* one *CPU*s, *and* one chunk of *1GB* of *RAM*, *and* one chunk of *10GB* of *HD*, *and* one gcc compiler. If a request succeeds, then, using a simple orchestration language, the *AC* client will use all resources offered by the servers *AC*s.

The *RDP* protocol proceeds as follows: suppose an *AC* X registers – using the intermittent protocol *VIP* presented below – to an *AB* and declares its availability to offer a service S, while another *AC* Y, already registered, issues a request for a service S'. Then, the *AB* looks in its *routing table* and *filters* S' against S. If there exists a solution to this filter equation, then X can provide a resource to Y. For example, the resource $\mathsf{S} \triangleq [\mathsf{CPU} = \mathsf{Intel}, \mathsf{Time} \leq 10\mathsf{sec}]$ filters against $\mathsf{S}' \triangleq [\mathsf{CPU} = \mathsf{Intel}, \mathsf{Time} \geq 5\mathsf{sec}]$, with attribute values Intel and Time between 5 and 10 seconds.

Routing tables in RDP. In *Arigatoni*, each *AB* maintains a *routing table* T locating the *services* that are registered in its colony. The table is updated according to the *dynamic registration and unregistration* of *AC*s in the overlay; thus, each *AB* maintains a partition of the data space. When an *AC* asks for a resource (service request), then the query is *filtered* against the routing tables of the *AB*s where the query is arrived and the *AC* is registered; in case of a *filter-failure*, the *AB*s forward the query to their direct super-*AB*s. Any answer of the query must follow the reverse path.

Thus, resource lookup overhead reduces when a query is satisfied in the current colony. Most structured overlays guarantee lookup operations that are logarithmic in the number of nodes. To improve routing performance, caching and replication of data and search paths can be adopted. Replication also improves load balancing, fault tolerance, and the durability of data items.

3.3.4. Virtual Intermittent Protocol (VIP)

There are essentially two ways *AC* can register to an *AB* (sensible to its physical position in the network topology), the latter being not enforced by the *Arigatoni* model (see [5]):

1. Registration of an *AC* to an *AB* belonging to the same *current administrative domain*;
2. Registration via *tunneling* of an *AC* to another *AB* belonging to a *different administrative domain*.

If both registrations apply, the *AC* is *de facto* working in local mode *in the current administrative domain* and working in global mode *in another administrative domain*. Symmetrically, an *AC* can unregister according to the following simple rules *"d'étiquette"*:

- Unregistration of an *AC* is allowed only when there are no pending services demanded or requested to the leader *AB* of the colony: agent computers must always wait for an answer of the *AB* or for a direct connection of the *AC* requesting or offering the promised service, or wait for an internal timeout (the time-frame must be negotiated with the *AB*);
- (As a corollary of the above) an *AB* cannot unregister from its own colony, *i.e.* it cannot discharge itself. However, for fault tolerance purposes, an *AB* can be faulty. In that case, the *AC*s unregister one after the other and the colony disappear;
- Once an *AC* has been disconnected from a colony belonging to any administrative domain, it can physically migrate in another colony belonging to any other administrative domain;
- Selfish agents in *P2P* networks, called "free riders", that only utilize other peers' resources without providing any contribution in return, can be fired by a leader; if the leader of a colony finds that the agent's ratio of fairness is too small ($\leq \epsilon$ for a given ϵ), it can arbitrarily decide to fire that agent without notice. Here, the *VIP* protocol also checks that the agent has no pending services to offer, or that the timeout of some promised services has expired, the latter case means that the free rider promised some services but finally did not provide any service at all (not trustfulness).

Registration policies in VIP. *VIP* registration policies are usually not specified in the protocol itself; thus every agent broker is free to choose its acceptance policy. This induces different self-organization policies and allow to reason on colony's load-balancing and kind of colonies. Possible politics and are:

- **(mono-thematic)** An agent broker accept an agent in its colony if the latter offer resources S that the colony already have in quantity $\geq \epsilon$, for a given ϵ;
- **(multi-thematic)** An agent broker accept an agent if the latter offer resources that the colony have in quantity $\leq \epsilon$, for a given ϵ;
- **(unbalanced)** An agent broker accept an agent always;
- **(pay-per-service)** An agent broker accept only agents that accept to pay some services;
- **(metropolis/village)** An agent broker accept an agent in its colony only if the number of citizens is greater/lesser than N;
- **(custom)** An agent broker accept an agent following a mix of the above politics.

3.3.5. Two simple examples

To give an idea of possible usage of the *Arigatoni* generic overlay network we present two examples; the first one has a *Grid*-computing flavor while the second is a nice interweaving of the *Arigatoni* overlay seated on the top of both *IP* and *MANET* underlay network. For more information the interested reader can have a look on [1] [20], [22].

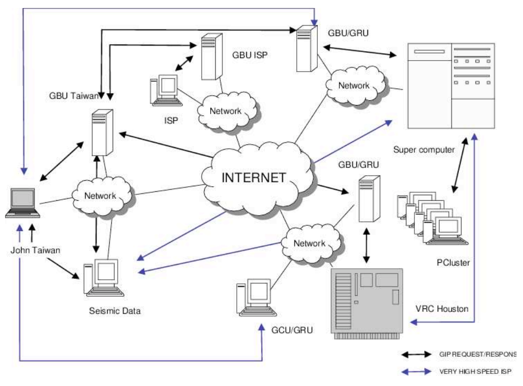


Figure 3. Arigatoni Overlay Network for a Grid Seismic Monitoring Application

GRID: scenario for seismic monitoring. John, chief engineer of the SeismicDataCorp Company, Taiwan, on board of the seismic data collector ship, has to decide on the next data collect campaign. For this he would like to process the 100 TeraBytes of seismic data that have been recorded on the data mass recorder located in the offshore data repository of the company to be processed and then analyzed. He has written the processing program for modeling and visualizing the seismic cube using some *parallel library* like *e.g.* `MPI` or `PVM`: his program can be distributed over different machines that will compute a chunk of the whole calculus; however, the amount of computation is so big that a supercomputer and a cluster of PC has to be *rented* by the SeismicDataCorp company. John will ask also for *bandwidth* in order to get rid of any bottleneck related to the big amount of data to be transferred. Then, the processed data should be analyzed using the *Virtual Reality Center, (VRC)* based in Houston, U.S.A. by a specialist team and the resulting recommendations for the next data collect campaign have to be sent to John. As such:

1. John logs on the *Arigatoni* Overlay Network in a given colony in Taiwan, and sends a quite complicated service request in order for the data to be processed using his own code. Usually the *AB* leader of the colony will receive and process the request;
2. If the Resource Discovery performed by the *AB* succeeds, *i.e.* a supercomputer and a cluster and an *ISP* are found, then the data are transferred at a very high speed and the *“Sinfonia”* begins;
3. John will also ask (in the *RDP* request) to the *AC* containing the seismic data to dispatch suitable chunks of data to the supercomputer and the cluster designated by the *AB* to perform some pieces of computation;
4. John will also ask (in the *RDP* request) to the supercomputer the task of collecting all intermediate results so calculating the final result of the computation, like a *“Maestro di Orchestra”*;
5. The processed data are then sent from the supercomputer, via the high speed *ISP*, to the Houston center for being visualized and analyzed;
6. Finally, the specialist team’s recommendations will be sent to John’s laptop.

This scenario is pictorially presented in Figure 3 (we suppose a number of sub-colonies with related leaders *AB*, all registered as agents to a super-*AB*;for example the John’s *AB* could be elected as the super-leader). For simplify security issues, all *AB*’s are trusted using the same *PKI*, making *de facto* in common all resources of their colonies. An animation of the coordination program, written in the visual language *JOpera* can be downloaded at http://www-sop.inria.fr/members/Luigi.Liquori/ARIGATONI/arigatoni_animation.wmv.

3.4. Background 2: SmartTools service-oriented overlay software factory

With the increasing dependency on the Internet and the proliferation of new component and distributive technologies, the design and implementation of complex applications must take into account standards, code distribution, deployment of components and reuse of business logic. To cope with these changes, applications need to be more open, adaptable and capable of evolving. To accommodate to these new challenges, Didier Parigot introduced in 2000 a new development approach based on generators associated with domain-specific languages, each of the latter related to one possible concern useful when building an application. It relies on Generative Programming, Component Programming and Aspect-Oriented Programming. A software factory, called *SmartTools* [44], [34], [35], [33], has been developed using this new approach. The main objective of this well established research vein are $i)$ to build software of better quality and to enable rapid development due to Generative Programming and, $ii)$ to facilitate insertion of new facets and the portability of applications to new technologies or platforms due to business logic and technology separation. In a nutshell:

- The *SmartTools*' *Service-Oriented Architecture* (*SOA*), is based on a set of adaptable components helping the construction of component-based application;
- On the conceptual level, *SmartTools* is based on the *Domain-Specific Language* (*DSL*) approach to model the various concerns;
- On the implementation level, *SmartTools* use techniques of *Generative Programming*.

3.4.1. SmartTools units

The important units of the *SmartTools* service oriented software factory are:

- A *Component Description Meta-Language* (*CDML*) for describing services of component in a neutral format (*XML*);
- A *Component Generator* (*CG*) that from *CDML* description generates the component's containers;
- A *Component Manager* (*CM*) controlling the life-cycle of components (loads, activations, etc.).

The main features of the *SmartTools* software factory are that

- The communications between components are based on asynchronous message passing;
- Each component is executed in a thread;
- The data are exchanged between components with a neutral *XML* format;
- The connections between components are established by the *CM* with an *Activation protocol*;
- Dynamic services are supported.

It is worth to notice that the role of a *CM* shares some similarities with the role of an *Arigatoni*'s *AB*. The *CM* controls a set of components and manage *service tables* available in every colony. A component demand access to a particular service through this Component Manager. In contrast to *Arigatoni*, the *SOA* specify, for each resource, all the methods (pieces of code implementing services. This defines the execution model between components (point-to-point).

3.4.2. SmartTools into Eclipse

Recently [45], [48], the *SmartTools* software factory was ported into the *Eclipse* environment, the latter being based on the *OSGi* framework. The port is available since October 2007 in the *SmartTools* V1.2 software release. In [37], we have studied the benefits of our service-oriented architecture on top of an *OSGi*-based *Eclipse* framework. To better understand the reasons of this choice, we quickly introduce *Eclipse* and *OSGi*.
Eclipse - an open development platform. *Eclipse* is an open source community whose projects are focused on building an open development platform including extensible frameworks, tools and run-time for building, deploying and managing software across the life-cycle. A large and vibrant ecosystem of major technology vendors, start-ups, universities, research institutions, complement and support the *Eclipse* platform.

OSGi - a dynamic module system for Java. The *OSGi* technology, is a component integration platform with a service-oriented architecture and life cycle capabilities that enable dynamic delivery of services. These capabilities greatly increase the value of a wide range of computers and devices that use the *Java* platform. The *OSGi* specifications provide the platform for universal middleware. The *OSGi* Alliance is a worldwide consortium of technology innovators that advances a proved and mature process to assure interoperability of applications and services based on its component integration platform. The *OSGi* Service Platform is delivered in many Fortune Agent 100 company products and services and in diverse markets including enterprise, mobile, home, telematics and consumer.

3.4.3. The SmartTools' evolution

As we said before, the current version of *SmartTools* V1.2 do not take into account distributed components nor overlay network issues. In other words, it is not suitable for a software factory of programmable overlay networks. Next releases of the new *SmartTools* factory will take into account this kind of applications.

To do this, we plan to expand the service-oriented architecture with service discovery and virtual service organizations. More precisely, the new *SmartTools* architecture will scale-up to a virtual organization of several service-oriented components, related to each others with a component manager playing the same role of an *Arigatoni*'s super-broker and organized in a *virtual service organization*. Registration between (distributed) service-oriented components will be featured on the basis of the same logic of the *Arigatoni*'s *VIP* protocol. Analogously, service discovery will be featured on the basis of the same logic of the *Arigatoni*'s *RDP* protocol.

3.4.4. The new SmartTools' functional units

We briefly presents the functional units of the new *SmartTools overlay software factory*.
Component Service (CS). This unit is the basic computational entity of the virtual service organization. A component service implements all services associated with a resource. It is an autonomous entity. Every *CS* register the list of services it can offer to the component manager described below.
Component Manager (CM). This unit represent the leader of the virtual service organization. *CM*s are organized hierarchically, as presented in Figure 4 (right).
Component Router (CR). This unit is the basic unit close to *CM*s and *CS*s that is devoted to sending and receiving packets of the service registration, the services discovery and the activation protocols. The connection *CM-CR* is ensured via a suitable *API*. Figure 4 gives an overview of the actual *SOA* (left) and the new one (right).

Very recently, a small experiment of making a flat organization virtual services has been done in the night-build distribution of *SmartTools* V1.2.1. In [43], we clearly show the need for a service discovery protocol, a virtual intermittent protocol, and an *ad hoc* activation and communication protocols. By using the *R-OSGi* library, a virtual organization (*i.e.* several *CM*s running on different Java Virtual Machines) has been partially validated with a very simple application. The *R-OSGi* library is based on a *Service Location Protocol* (*SLP*) used for service discovery and a *Message Oriented Middleware* for communication between *CS*s based on asynchronous messages. This first experience showed that a hierarchical organization is useful to avoid an overload of communication on each colony.

3.5. General research directions

Following our main three topics, network organization, resource discovery and orchestration, *Arigatoni* and *SmartTools* share the same concepts but are complementary: *Arigatoni* goes deeper in the network organization and the resource discovery (*VIP* and *RDP*) while *SmartTools* concentrates in orchestration.

Thus the short term project will combine in a same framework *Arigatoni* and *SmartTools*. We will make an implementation of the *VIP* and *RDP* protocols which can be used by *SmartTools* bringing to *Arigatoni* a way to resource orchestration. For middle and long term research, we envisage the following studies.

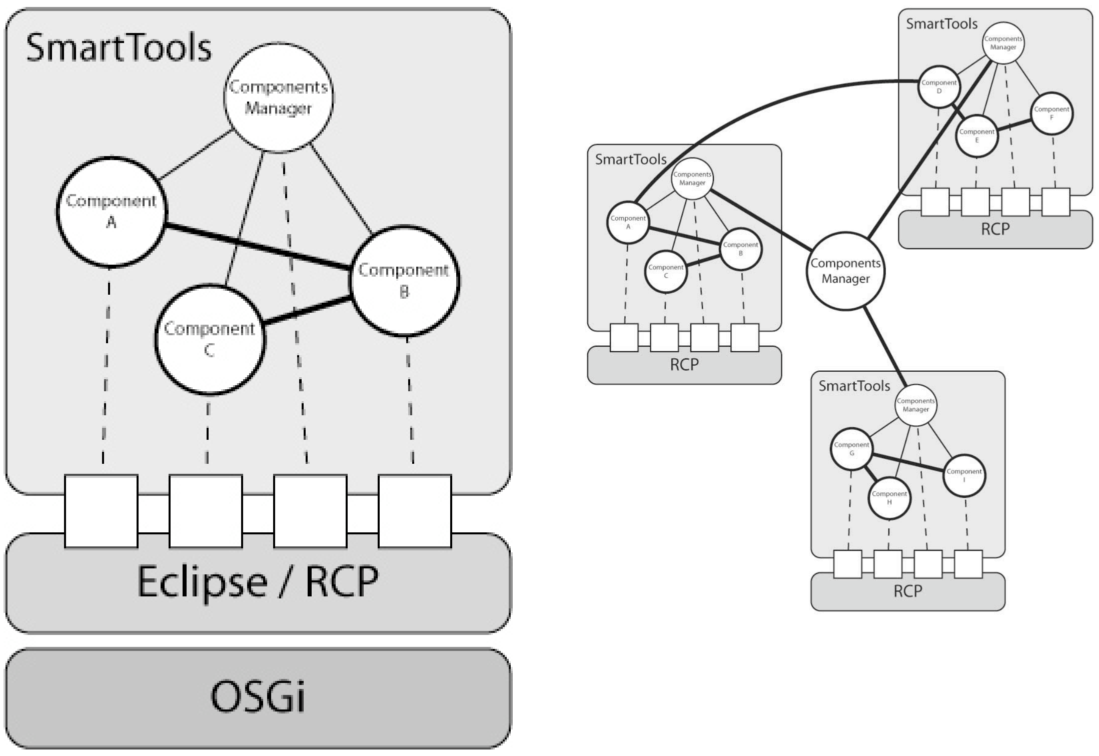


Figure 4. SmartTools on top of Eclipse OSGi Framework and the new virtual service organization of SmartTools

3.5.1. On Virtual organizations

- **Trees *vs.* graphs: a conflict without a cause**. In the first versions of *Arigatoni*, the network topology was tree- or forest-based. But since agents are not slaves, multiple registrations are in principle possible and unavoidable. This weaves the network topology into a *dynamic graph* [38], where nodes do not have a complete knowledge of the topology itself. As an immediate consequence, our protocols must deal with multiple registrations of the same agent in different colonies, with the natural consequence of resource overbooking, routing table update loops (when a service update request comes back to the broker that generates the request itself), and resource discovery loops (when a resource service request comes back to the agent that generates the request itself), see [10].

  As an example of resource overbooking, suppose an agent computer registers to two colonies, by declaring and offering the same resource S twice, *i.e.* once for each colony. This phenomenon is well known in the telecommunications industry, as in the “frame-relay” world. For the record, overbooking in telecommunications means that a telephone company has sold access to too many customers who basically flood the telephone company lines, resulting in an inability for some customers to use what they purchased. Other examples of overbooking can be found in the domain of transportation (airlines) and hotel reservations.

  Resource discovery is a non-trivial problem for large distributed systems featuring a discontinuous amount of resources offered by agent computers and their intermittent participation in the overlay. Peers’ intermittence lead also to design new routing algorithms and protocols stable to agent churn; this scenario can modeled using dynamic graph theory.

- **Fault tolerance**. The virtual organization model offers some mechanisms to dynamically adapt to *dynamic topology changes* of the overlay network, by allowing an agent (computer or broker, representing a sub-colony) to login/logout in/from a colony. This essentially means that the process of routing requests and responses may lead to failure, because some agents logged out or because they are temporarily unavailable (recall that agents are not slaves). This may also lead to temporary denials of service or, more drastically, to the complete “delogging” of an agent from a given colony in the case where the former does not provide enough services to the latter.

3.5.2. On Resource discovery

- **Parametricity and universality**. Dealing only with resource discovery has one important advantage: the complete generality and independence of any offered and requested resource. Thus, *Arigatoni* can fit with various scenarios in the agent computing arena, from classical *P2P* applications, like file- or band-sharing, to more sophisticated *Grid* applications, like remote and distributed big (and small) computations, until possible, futuristic *migration computations*, *i.e.* transfer of a non completed local run in another agent computer, the latter being useful in case of catastrophic scenarios, like fire, terrorist attack, earthquake, etc., in the vein of agent programming languages *à la Obliq* or *Telescript*. We could envisage at least the following scenarios to be a tight fit for our model:
  - Ask for computational power (*i.e.* the *Grid*);
  - Ask for memory space (*i.e.* distributed storage);
  - Ask for bandwidth (*i.e. VoIP*);
  - Ask for a distributed file retrieving (*i.e.* standard *P2P* applications);
  - Ask for a (possibly) distributed web service (*i.e.* query *à la Google or any service available via web-oriented protocols);*
  - *Orchestration of a distributed execution of an algorithm (i.e. a kind of distributed von Neumann machine);*
  - *Ask for a computation migration (i.e. transfer one partial run in another agent computer, saving the partial results, as in a truly mobile ubiquitous computation);*

  - *Ask for a human computer interaction (the human playing the role of an agent)...*

- **Social model underneath an overlay network computer**. The *Arigatoni* overlay network computer defines mechanisms for devices to inter-operate, by offering services, in a way that is reminiscent to Rapoport's *tit-for-tat* strategy of co-operation based on reciprocity. This way to understand common behavior of virtual organizations has some theoretical basis on Game Theory. Classical results from game theory are based on the assumption that a shared amount of resources is available and then users have an incentive to collaborate. The very first design of *Arigatoni* forced each *AC* to register to only one *AB*. But, recent studies showed that the *Arigatoni* overlay can be smoothly scaled up to a more general topology where each *AC* may simultaneously be registered to several *AB*, and where a *colony* is just one possible *social scheme* [28].

  This means that *Arigatoni* fits with motivations and cooperation behavior of different communities. It tries to be *policy neutral*, leaving policy choices for each agent at the implementation or configuration level, or at the community or organization level. Policy domains can overlap (one agent can define itself as belonging "much" to colony *foo* and "a little bit" to colony *bar*). This denotes a decentralized non-exclusive policy model. As such, one question can arise: who is *Arigatoni* designed for? We believe the overlay is flexible enough to serve a mix of different "social structures" and "end-users":

  - Independent end-user connecting through his *ISP* or migrating from hot-spot to hot-spot;
  - Cooperative communities of disseminated agents;
  - More regulated or hierarchical communities (maybe a better view of a corporate network);
  - Cooperative or competitive resource providers and resource brokers.

- **Quality metrics underneath an overlay network computer**. The *Arigatoni* overlay network computer is suitable to support various extended trust models. Moreover, reputation score could be expanded to a multi-dimensional value, for example adding a score for quality of the service offered by an agent. However, *Arigatoni* encourages cooperation and enables gratuitous resource offering. But it may also suit for business extensions, *e.g.*:

  - An agent computer can sell resource usage, creating a resource business;
  - An agent broker can sell a resource discovery service, creating a brokering business (*"I point you to the best resources, more quickly than anyone else"*).

  The *Arigatoni* overlay network computer is suitable of a number of service extensions: among others:

  - How to create and call third party services for on-line payment of services;
  - How to exchange digital cash for payment of services;
  - How to negotiate service conditions between client and servants, including price and quality of service.

  The one-to-many nature of the *RDP* protocol service request (*SREQ*) are of particular interest in this case. Another possible *Arigatoni* extension may define how to join a third party auction server. Candidate servants for a *SREQ* would contact the auction server and make their bid. The trusted auction server chooses the elected candidate and service conditions based on auction terms. The agent would then contact the auction server and get this information. Those extensions may take advantage of the *RDP* optional fields [1], for example to transmit location and parameter information to call a third party system.

3.5.3. Execution model

- **Programming an overlay network computer**. Once resources (hardware, software...) have been discovered, the agent computer that made the request may wish to use and manipulate it; to do this, the agent computer has written a (distributed) program in a new language (*à la BPEL*, *LINDA*, *YAWL*, *JOpera*...), let's call it *Ivonne*, in honor to the great scientist John von Neumann.

Those languages are often called (terminology often overlaps), *coordination- workflow- dataflow- orchestration- composition- metaprogramming- languages*. *Ivonne* will have *ad hoc* primitives to express sequences, iterators, cycles, parallel split, joins, synchronization, exclusive/multi/deferred choice, simple/multi/synchronizing merge, discriminators, pipelining, cancellation, implicit termination, exception handling... [49].

The "main" of an *Ivonne* program will be runned on the agent computer machine that launched the service request and received the resources availability: it will orchestrates and coordinates data and program resources executed on others agent computers.

In case of *failure* of a remote service – due to a network problem or simply because of the unreliability or untrustability of the agent that promised the resource – an exception handling mechanism will send a resource discovery query *on the fly* to recover a faulty peer and the actual state of the run represented, in semantic jargon, by the *current continuation*.

We also envisage to design a *run-time* distributed virtual machine, built on the top of a virtual or hardware machine, in order to scale-up from local to distributed computations and to fit with the distributed nature of an overlay network computer. Communication between agent computers will performed thru a *logic bus*, using Web technologies, like *SOAP* or *AJAX* protocols, or a combination of *Java*-based *JNI+RMI*-protocols, or *.NET*, *XPCOM*, *D-BUS*, *OLE* bus protocols, or even by enriching the *Arigatoni* protocol suite with an *ad hoc* control-flow and data-flow protocol, and permitting to use it directly inside *Ivonne*.

The *Ivonne* language can be both interpreted or compiled. In the latter case we envisage the design of an intermediate low-level distributed assembler language where *Ivonne* could be compiled. The intermediate machine code will recast the assembler pseudo code
`move R0 R1`
à la Backus [27] in
`move dataR0 from ipR0:portR0 to ipR1:portR1`
where of course latency is an non-trivial issue, or the assembler pseudo code
`op R0 R1 R2` in
`op on ipR0 with ipR0:portR0:dataR0 and ipR1:portR1:dataR1 and`
`stockin ipR2:portR2:dataR2`.
Resuming, an *overlay program* will be a smooth combination of an overlay network connectivity dealing with virtual organizations and discovery protocols, a computation of an algorithm resulting of the *summa* of all algorithms running on different computer agents, and the coordination of all computer agents, made by an *Ivonne* program.

- **Programming with** *SmartTools*' *SOA*

  Using our activation protocol of services, *SmartTools SOA* provides an execution model (programmable model) on a flat organization.

  To achieve the goal of "Programmable Overlay Network", it will be necessary that *SmartTools* component manager use directly the *VIP* and *RDP* protocols. Moreover, the activation protocol (discovery and launch *SmartTools* components) and the transport (communication) protocol will have to be adapted to this new virtual organization. To facilitate future evolutions, this new version of our *SOA* will be based on the definition of our own protocols (activation and transport) without complicated libraries (as *R-OSGi*).

  All of these research elements will allow to develop a concept of dynamic *SOA* to meet the demands of applications on such virtual organization.

  Our approach is characterized by a dynamic topology, and *P2P* communication between components. The classical concept of workflow induces often a static topology of services and a centralized orchestration. Therefore, we want to introduce a new concept of dynamics and decentralized orchestration. When carrying relationship of resources (components), the constraints of use should allow to define the mechanisms (forced) of communications, which define locally the workflow. Finally, the global workflow of the application results from the combination of local workflow.

However this solution must be able to integrate or collaborated solutions with more conventional orchestration. Indeed it can be assumed that a orchestration with large grains will finally be useful.

- **Trust and security**. In order to work securely, the *Arigatoni* overlay network computer needs to be able to offer the following guarantees to its components:
  - The communication between two agents must be secured;
  - The role played by an agent (*i.e.* client *AC*, servant *AC* or *AB*) must be certified by a third party trusted by the agents that communicate with this particular agent. A way to implement those constraints is to use *PKI* certificates. A *Certification Authority* delivers certificates, and couples of private and public keys for *AC*s and *AB*s which attest of their distinctive roles. The whole mechanisms involved by a *PKI* is out of the scope of this research statement, but good use of *PKI*s and an implementation compliant with *RFC2743* can provide all the necessary security, namely the trustfulness on the identity of the peers, and the trustfulness of all the transmitted data, *i.e.* secrecy, authenticity, and integrity;
  - In addition to *PKI*s, a more “liquid” trust model could be built, based on *reputation* mechanisms. Reputation represents the amount of trust an agent in the overlay has in another agent based on its partial view. In a nutshell:
    - Each agent maintains a reputation score for each agent it knows;
    - Each agent maintains a reputation score for each resource it serves;
    - Exchanges between agents update dynamically each other’s scores;
    - Conflict between two or many agents are resolved by the brokers leaders of the colonies to which agents belong;
    - The computation of the reputation score (a trust metrics) and the way agents exchange scores is left free to each single implementation.

  A last word on implementation issues of the *Arigatoni* overlay network computer: it is well known that two technical barriers are commonly used to block transmission over *IP* network in overlays:
  - *Firewalls* to drop *UDP* flows (usually considered as suspects);
  - *NAT* techniques to mask to the outside world the real *IP* addresses of inside hosts; a *NAT* equipment changes the *IP* source address when a packet *goes to* outside, and it changes the *IP* destination address when a packet *comes from* outside.

  The usage of these mechanisms is very frequent on the Internet and they are barriers that can prevent connections between *inside* and *outside* agents in *Arigatoni*. The implementation of *RFC3489* could be used to overcome such obstacles.

4. Application Domains

4.1. Panorama

Because of its generality, our overlay network can target many applications. We would like to list a small list of useful programmable overlay networks case of study that can be considered as “LogNet Grand Challenges” to help potential readers to understand the interest of our research program.

- New distributed models of computation
- Overlay networks over mobile *ad hoc* networks
- Reduce the digital divide

4.2. Potential applications

Keywords: *New distributed models of computation*, *mobile ad hoc networks*, *overlay networks*, *reduce the digital divide*.

From large-scale computing machines to large-scale overlay network machines (John von Neumann was right before all). This challenge is inspired by the seminal talk by John von Neumann, given in May 1946, “Principles of Large-Scale Computing Machines”, typesetted and reprinted in [50]. At that time, “large-scale” meant the *ENIAC* computer, *i.e.*, 17,468 vacuum tubes, 7,200 crystal diodes, 1,500 relays, 70,000 resistors, 10,000 capacitors, 5 million joint, 30 short tons, 2.4m x 0.9m x 30m, stored in a 167 m^2 room, and 150 kW to operate. Today, thanks to the Moore’s law and to the Internet, “large scale” means “worldwide scale”, *i.e.* the computer hardware is distributed in space and in time and must be negotiated before being used. The main inspirations of the programmable overlay network computer research’s vein are still contained in that article.

The term “von Neumann bottleneck” was coined by John Backus in his 1977 ACM Turing award lecture. Bottleneck refers to the fact that, since data and program are stored on the same support (the memory), the throughput (data transfer rate) between the CPU and the memory is very low. In current von Neumann architecture the bottleneck is alleviated by using big cache memories. Since in overlay network computers the bus can be modeled by an Internet connection, the data transfer is still more critical than on a single processor machine. As such, we should probably look at new computer architectures, such as the Harvard one.

Needless to say that the “icing on the cake‘” will be to formalize this new distributed computational model and architecture, together with a formal proof of its Turing completeness statement!
Developing a pedestrian/vehicular infrastructure based on an overlay network computer. We plan to build an *ad hoc* vehicular network infrastructure using the *Arigatoni* overlay infrastructure. That network must enable efficient and transparent access to the resources of on-board and roadside agents. In such a scenario, commercial services and access to public information are available to vehicles transiting in specific areas where such information is broadcast by roadside wireless gateways or by other vehicles. Data retrieved can be stored on the on-board vehicle computer; then, they can be used and rebroadcast at a later time without the need of persistent connectivity. These new features will offer innovative functions and services, such as:

- Distribution, from infrastructure to vehicle (*I2V*), and among vehicles (*V2V*), of safety and/or traffic-related information;
- Collection, from vehicles to infrastructures (*V2I*), of data useful to perform traffic management;
- Exchange of information between private vehicles and public transportation systems (buses, vehicles, road side equipments...) to support and, thus, foster inter-modality in urban areas;
- Distribution of real-time, updated information to enable dynamic navigation services.

In this scenario, vehicles/pedestrians play the role of agent computers, while Bus-stop stations equipped with *IP* network, routing tables and *WIFI* access point play the role of agent brokers; Buses play the role of mobile agent brokers, a sort of proxy of a unique bus-stop agent broker. Proxy load balancing policies are left to the bus headquarter (*HQ*). See, for more details, the *Arigatoni*’s sub-project *Ariwheels*.
Programming services for the new mesh overlay network in the Campus STIC of Sophia Antipolis. The future Campus STIC, grouping EPU, UNSA, Eurecom, CNRS, and INRIA will be ready in one year. It will be equipped with a *WIFI* network infrastructure implementing 802.11a/b/g protocols, with potential evolution to 802.11n protocol. The main objectives of such underlay network are to offer *IP* connection to all Campus “citizens”: the network must guarantee the respect of French laws concerning public network connections (*décret 2006-358 sur l’offre de connexion au public loi 2006-64*). To do this, it would be suitable that all users get identified using, *e.g.*, using the “pin” code of the student/employee-card. The infrastructure mainly targets Internet access for all. The Campus STIC *WIFI* underlay network could be an unique opportunity to have a real testbed into which put our programmable overlay at work. *Arigatoni* and *Ariwheels* could represent the overlay network infrastructure to offer *much more* than simply an Internet connection: the LogNet vision can provide a list of interesting high-level semantic (on demand) services, and a plausible way to implement it.

Reducing the *Digital Divide* [Sources Wikipedia]. The digital divide is the troubling gap between those who use computers and the Internet and those who do not. The term digital divide had a moving target: first, it meant the ownership of a computer. Later and access to the Internet. Most recently it centers on broadband access. In modern usage, the term also means more than just access to hardware, it also refers to the imbalance that exists amongst groups of society regarding their ability to use information technology.

The digital divide tends to focus on access to hardware, access to the Internet. The writer Lisa J. Servon argued in 2002 that the digital divide "is a symptom of a larger and more complex problem – the problem of persistent poverty and inequality". The four major components that contribute to digital divide are "socioeconomic status, with income, educational level, and race among other factors associated with technological attainment".

One area of significant focus was *school computer access*; in the 1990s, rich schools were much more likely to provide their students with regular computer access. In the late 1990s, rich schools were much more likely to have Internet access. In the context of schools, which has constantly been involved in the discussion of the divide, current formulations of the divide focus more on how (and whether) computers are used by students, and less on whether there are computers or Internet connections.

The USA E-rate program (officially the Schools and Libraries Program of the Universal Service Fund), authorized in 1996 and implemented in 1997, directly addressed the technology gap between rich and poor schools by allocating money from telecommunications taxes to poor schools without technology resources. Though the program faced criticism and controversy in its methods of disbursement, it did provide over 100,000 schools with additional computing resources, and Internet connectivity.

Recently, discussions of a digital divide in school access have broadened to include technology related skills and training in addition to basic access to computers and Internet access. An interesting example is that, in the North of Italy, the town of Pordenone, 50,000 inhabitants, will be equipped with public local *WIFI* LAN (*e.g.* see the declaration of the Major, in Italian, http://it.youtube.com/watch?v=zBTnkEnXTlc). Our vision could contribute to reduce digital divide in our society and more contextually in the future Campus STIC.

5. Software

5.1. Ariwheels

Keywords: *Mobile ad hoc networks*, *content-based networks*.

Participants: Luigi Liquori [contact for the *Ariwheels* simulator], Claudio Casetti [Politecnico di Torino, Italy], Diego Borsetti [Politecnico di Torino, Italy], Carla-Fabiana Chiasserini [Politecnico di Torino, Italy], Diego Malandrino [Politecnico di Torino, Italy, contact for the *Ariwheels* client].

Ariwheels is an infomobility solution for urban environments, with access points deployed at both bus stops (forming thus a wired backbone) and inside buses themselves. Such a network is meant to provide connectivity and services to the users of the public transport system, allowing them to exchange services, resources and information through their mobile devices. *Ariwheels* is both:

- a protocol, based on *Arigatoni* and the publish/subscribe paradigm;
- a set of applications, implementing the protocol on the different types of nodes;
- a simulator, written in OMNET++ and recently ported to the ns2 simulator.

5.1.1. Simulator

We implemented *Ariwheels* within the Omnet++ simulator, coding the overlay part and exploiting the existing wireless underlay network modules. In the underlay, we used IEEE 802.11 at the MAC layer and the DYMO routing protocol (an AODV-like reactive routing protocol).

We tested the performance of *Ariwheels* in a vehicular environment. We used a realistic mobility model generated by the simulator VanetMobiSim, whose output (mobility traces) was fed to the Omnet++ simulator. Vehicles travel in a 1 km^2 city section over a set of urban roads, which include several road intersections regulated by traffic lights or stop signs. In particular, we adopted the IDM-IM microscopic car-following model [39], which allows us to reproduce real-world traffic dynamics as queues of vehicles decelerating and/or coming to a full stop near crowded intersections.

We assumed that 60 vehicles enter the city section from one of the border entry/exit points, randomly choose another border entry/exit point as their destination, compute the fastest path to it and then cross the city section accordingly. A vehicle entering the topology is assigned a top speed of v m/s, that it tries to reach and maintain, as long as traffic conditions and road signs allow it to. When a vehicle reaches its destination, it stops for a random amount of time, uniformly distributed between 0 and 60 s, then it re-enters the city section. In our simulations, we tested two different top speeds v: 9 m/s (approx. 32 km/s) and 15 m/s (approx. 54 km/s).

Upon entering the topology, a vehicle acting as Mobile Agent owns a set of 12 unitary services (*e.g.*, files, traffic informations, point of interests) randomly chosen from a set of 20 services. A Mobile Agent issues a (*SREQ*) for a service it is missing and the inter-request time is supposed to be exponentially distributed with parameter $\lambda = 0.05$ [req./s]. As typical in the publish/subscribe paradigm, where peers are not slaves, upon receiving a *SREQ* for a service it owns, a Mobile Agent sends back a positive response with a certain probability, which is set to 0.9 in our simulations.

The simulated city topology, features 6 bus stops with *AP*s, each corresponding to a Broker. Furthermore, 3 buses acting as Mobile Brokers weave their own routes across the topology, among a population of as many as 60 vehicles acting as Mobile Agents. Each bus carries 10 passengers equipped with Mobile Agent capabilities, and it associates to the Broker with the smallest colony at the time of departure from the bus station. Brokers apply the unbalanced acceptance policy and filter the routing table against a received query by using the liveliness information only.

5.1.2. Network client

Scenario. *Ariwheels* is designed for the scenario of urban public transportation. In such a scenario, a significant number of users equipped with mobile devices spends significant amounts of time at the bus stops or inside the bus themselves. The basic idea of *Ariwheels* is to exploit this situation to let the users exchange data or services - more generally: resources - through their mobile devices.
Infrastructure. *Ariwheels* is based on the 802.11 wireless LAN protocols. Therefore, its infrastructure is mostly made of access points, deployed both:

- at bus stops, forming a backbone;
- on the bus themselves, thus with an intermittent connection to the backbone.

The infrastructure also includes the IP network connecting the coverage areas of the access points, and some higher-level coordination facilities. Nodes and software

Network nodes will be quite ubiquitous, including bus stops, buses and passengers, *i.e.* the ones equipped with a suitable mobile device. Most nodes will be mobile (*i.e.* they move) and dynamic (*i.e.* they can suddenly be turned off, or leave the network itself). Owing to the peculiarities of its nodes, the *Ariwheels* network falls in the category of mesh networks.

All nodes will run *ad hoc* pieces of software. In other words, installing the *Ariwheels* software makes the difference between a node of the underlay network (802.11, IP) and a node of the overlay *Ariwheels* network.
Solution. *Ariwheels* have four kinds of functional units, namely agents, brokers, mobile brokers, and proxies.

These units interact according to a protocol based on the publish/subscribe paradigm.

Agent. The agent is a software - written in C# for better compatibility - running on the user's device. It will run in user space and unprivileged mode, in order to require no additional configuration or permissions. Using appropriate sensing and probing mechanisms, the agent will look for a Broker. Once found one, it performs:

- the registration, which includes sending a list of the resources the agent has to offer;
- the request of the resources the agent needs.

Registration is performed only once - *i.e.* once every time the agents meets a broker. Resource requests are usually repeated several times, until all the needed resources are found. In addition to this seeking activity, the agent has to provide the services it claimed to be providing to the agents requesting them.
Broker. The broker is a program, written in C for better performance, running on a mid- or high-end device. There must be (at least) one broker in each L2 network belonging to the *Ariwheels* system. The Broker performs four main duties:

1. advertise its presence and the resources available through it;
2. receive, elaborate and acknowledge the registration requests coming from the Agents;
3. receive and (try to) answer the resource request coming from the Agents;
4. manage feedback and reputation.

Internally, the broker is equipped with an embedded database (namely SQLite), which is updated at each received packet. The information contained in the packets, as well as the information represented by the packet itself is combined in a (sort of) routing table. For each resource, it contains ID and *IP* address of the agent which will be asked to provide it. This agent will be chosen among the ones providing the resources according to a policy (hopefully) balancing availability and fairness. After identifying the agent which will have to provide the resource, the broker pings it, in order to avoid committing the supplying of the resource to an agent which is not active anymore.
Mobile broker. Mobile brokers are brokers with an intermittent connection to the rest of the network. A typical example is a bus equipped with a wireless access point, connecting - when possible - to the infrastructure, deployed at some stops. Mobile brokers are associated with a fixed broker. As soon as this broker becomes available (*i.e.* the mobile broker can hear its Hello messages), the mobile broker sends it one or more Dump messages, containing its routing table. The fixed broker replies dumping its own routing table. As a result, both mobile and fixed brokers know which services are available through the other. As long as the connection lasts, the two brokers will use this information to answer the service requests they cannot satisfy using their own routing tables. The priority order is:

1. the broker's own table;
2. the table dumped by mobile/fixed brokers;
3. forwarding the request to the proxy parent (see below), if any.

When the connection with a broker is lost, the routing entries relating to it are flushed.
Proxy. Proxies are the way *Ariwheels* copes with the need to access information outside the colony. The following basic principles hold:

- brokers only store information about their own colony;
- brokers are the only entity storing information.

As a consequence, there is no such thing as a super-broker, *i.e.* a node having global (or higher-level) information about the network. The rationale for this is that the rate at which the network changes is comparable to the time needed to propagate such information. Lacking super-brokers, agents still have the opportunity to consume services provided outside the colony they belong to. The Proxy node will handle the forwarding of *SREQ*'s and *SRESP*'s across colony borders, according to the schema:

1. brokers know in advance their parent proxy, and regularly send it Proxy packets;
2. if a broker is unable to answer a *SREQ* from one of its agents, it forwards it to its proxy;
3. the proxy forwards the *SREQ* to all its children brokers;
4. brokers reply to this forwarded *SREQ* only if they know how to gather the requested service;
5. if a *SRESP* arrives, the proxy forwards it to the broker having originated it;
6. otherwise, after a timeout, sends to the originating broker an empty *SRESP*;
7. the broker forwards the response it receives from the broker, either full or empty, to its agent.

Basic interaction. The most basic interaction between two agents and a broker foresees the following steps:

1. agent B registers with the broker, declaring to provide (among others) a given resource R;
2. agent A registers;
3. agent A queries the broker for resource R;
4. the broker looks up its routing table, and chooses B to provide R to A;
5. the broker transmits to A the details of B (basically, its IP address);
6. A contacts B, asking it for resource R;
7. B provides resource R to A.

Note that the actual data exchange between A and B happens in a fully peer-to-peer fashion, and does not involve the broker. Additionally, the broker itself does not provide any service: it only knows who could provide it. Advertising

Although not included in the basic interaction, the advertising mechanism - *i.e.* how does the Agent get to know that there is a Broker out there? has an important role. The solution adopted in *Ariwheels* is an Hello/Probe mechanism:

- the Brokers send, at random intervals, a Hello packet, which includes the list of the resources available through them;
- the Agents which are interested in looking for a broker (either because they know none, or because they want to change their current one) send a Probe packet;
- the Brokers receiving a Probe packet answer with a (unicast) Hello packet.

Advertising mechanisms usually foresee the usage of broadcast traffic. However, since *Ariwheels* is built upon IP, it can profit by the often neglected feature of multicasting. Hello and Probe packets are sent to distinct, well-known multicast addresses. The underlay network is expected to be configured in such a way that multicast packets are not forwarded outside the L2 network they originate in. As a result, only agents interested in knowing new brokers will join the Hello multicast group. Additionally, such a group will be also joined by brokers interested in knowing other brokers. See the web page http://www-sop.inria.fr/members/Luigi.Liquori/ARIGATONI/Ariwheels.htm and http://arigtt.altervista.org.

5.2. PiNet

Keywords: *Distributed Hash Table.*

Participant: Bernard Serpette [contact].

PiNet is a *Java* library (about 2K lines of code) whose main goal is to provide an abstract access to logical networks. Its specific features are:

- every value can be made available in a network via a dedicated node; compared to RMI, an available value does not need to inherit from a `Remote` class and there is no stub generation;
- each node has a unique thread of calculus (a sequencer), avoiding synchronized methods;
- provide functions to release the sequencer when blocking methods are needed;
- communications between nodes abstracts the transport layer, *i.e. TCP*, *UDP* or other protocols are not hard coded in the library;
- strong typing of messages exchanged between nodes, a channel established between two nodes is created to communicate values of a specific type;
- encourage the communication of functions, this feature has allowed the strong typing of channels, the abstraction of transport protocol and the absence of stub generation;
- provide some predefined overlay networks such as *Arigatoni* or Distributed Hash Tables (*DHT*).

5.3. ArchiNet

Keywords: *Service Oriented Architecture*.

Participants: Didier Parigot [contact], Baptiste Boussemart.

ArchiNet is a framework to build a *SOA* on the top of an Overlay Network. Its specific features are:

- a resource is described by its input or output (required or provided) services;
- a resource is associated with the component that implements its services;
- on each node (agent) an instance of *ArchiNet* is launched;
- each *ArchiNet* client is connected to a overlay network (as *PiNet*);
- each *ArchiNet* manages the resources available (creation of a component) on the node, resource available locally ;
- a resource available on a node is published on the overlay network by *ArchiNet*;
- *ArchiNet* asks a resource (component) to the overlay network if it does not exist locally;
- the sending or receiving services between components are transformed by asynchronous messages (by the components themselves);
- the local or remote connection between component (composition of services) is established by the *ArchiNet* associated to each component.

ArchiNet derives from the *SmartTools* software factory. This has induced the following transformations and operations.

- a better separation between *SOA* and *SmartTools* itself (the generation tools of plugins);
- best decomposition of the component manager (*ArchiNet*) to take into account different communication protocols.
- creation of communication module for *UDP* and *TCP* protocol
- a connection to an overlay network as *PiNet*.
- creation of some examples for tested *ArchiNet* connected to *PiNet*, such as a Chat in a *P2P* mode;
- test on different OS (Linux, Windows, Mac) and on smaller devices such as the Nokia N800;

See the web page http://www-sop.inria.fr/lognet/pon.

5.4. SmartTools

Keywords: *Domain-Specific Languages*, *Service Oriented Architecture*.

Participants: Didier Parigot [contact], Baptiste Boussemart.

The SmartTools software factory is a set of Domain-Specific Languages with associated tools used to develop more rapidly *Eclipse*-based application. The main advantages of this *SmartTools* approach are:

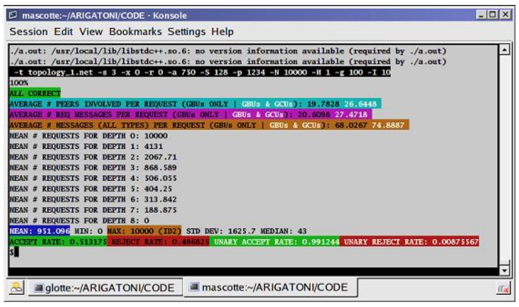
- A set of models or Domain Specific Language (*DSL*) drives your plugins software development. The *DSL* of your language can automatically produce the *Java* model (*Java*class to represented Abstract Syntax Tree), and the parser for a particular concrete syntax and various graphical views;
- A Service-Oriented Architecture (*SOA*) on top of the *Eclipse* framework for your composition plugins. Thank to particular *DSLs* (definition of the *OSGi* services), this *SOA* automatically manages the communications (with asynchronous messages) between your plugins;
- The *SmartTools* approach is completely integrated into the *Eclipse* environment. *SmartTools* is a complementary toolbox to the basic *Eclipse* framework.

The *SmartTools* software factory has been presented at *Eclipse* conference 2008 [21].

See the web page http://www-sop.inria.fr/lognet/SmartTools/eclipse and the INRIA GFORGE page http://gforge.inria.fr/projects/smarttools.

5.5. Arigatoni simulator

Participants: Luigi Liquori [contact], Raphael Chand [Université de Geneva, Switzerland].


mascotte:~/ARIGATONI/CODE - Konsole
Session Edit View Bookmarks Settings Help
glotte:~/ARIGATONI/CODE
mascotte:~/ARIGATONI/CODE

```
./a.out: /usr/local/lib/libstdc++.so.6: no version information available (required by ./a.out)
./a.out: /usr/local/lib/libstdc++.so.6: no version information available (required by ./a.out)
 -t topology_1.net -s 3 -x 0 -r 0 -a 750 -S 128 -p 1234 -N 10000 -M 1 -g 100 -I 10
100%
ALL CORRECT
AVERAGE # PEERS INVOLVED PER REQUEST (GBUs ONLY | GBUs & GCUs): 19.7828 26.6448
AVERAGE # REQ MESSAGES PER REQUEST (GBUs ONLY | GBUs & GCUs): 20.6098 27.4718
AVERAGE # MESSAGES (ALL TYPES) PER REQUEST (GBUs ONLY | GBUs & GCUs): 68.0267 74.8887
MEAN # REQUESTS FOR DEPTH 0: 10000
MEAN # REQUESTS FOR DEPTH 1: 4131
MEAN # REQUESTS FOR DEPTH 2: 2067.71
MEAN # REQUESTS FOR DEPTH 3: 868.589
MEAN # REQUESTS FOR DEPTH 4: 506.055
MEAN # REQUESTS FOR DEPTH 5: 404.25
MEAN # REQUESTS FOR DEPTH 6: 313.842
MEAN # REQUESTS FOR DEPTH 7: 188.875
MEAN # REQUESTS FOR DEPTH 8: 0
MEAN: 951.096 MIN: 0 MAX: 10000 (ID2) STD DEV: 1625.7 MEDIAN: 43
ACCEPT RATE: 0.513175 REJECT RATE: 0.486825 UNARY ACCEPT RATE: 0.991244 UNARY REJECT RATE: 0.00875567
$
```

Figure 5. The Arigatoni simulator

We have implemented in C++ ($\sim$2.5K lines of code) the Resource Discovery Algorithm and the Virtual Intermittent Protocol of the Arigatoni Overlay Network. The simulator was used to measure the load when we issued n service requests at Global Computers chosen uniformly at random. Each request contained a certain number of instances of one service, also chosen uniformly at random. Each service request was then handled by the Resource Discovery mechanism of Arigatoni networks.

5.6. BabelChord simulator

Participant: Cédric Tedeschi [contact].

To better capture its relevance, we have conducted some simulations of the BabelChord approach. The simulator, written in Python, works in two phases. First, a Babelchord topology is created, with the following properties: (i) a fixed network size (the number of nodes) N, (ii) a fixed number of floors denoted F, (iii) a fixed global *connectivity*, *i.e.*, the number of floors each node belongs to, denoted C. As a consequence: (i) The nodes are uniformly dispatched among the floors, *i.e.*, each node belongs to C floors uniformly chosen among the set of floors. (ii) Each resource provided by nodes is present at C floors. (iii) The average lookup length within one given floor is $\log((N \times C)/F)/2$.

In a second time, the simulator computes the number of hops required to reach one of the node storing one of the key of a particular resource. Results are given for different values of N, F, and C. Figure 6 shows the number of synapses vs. the lookup success rate. **Note that only 5% of synapses made of 2 (resp. 3, 5, 10) floors connections in the whole node population is enough to achieve more than 50% (resp. 60%, 80%, 95%) of exhaustive lookups in the Babelchord network!**

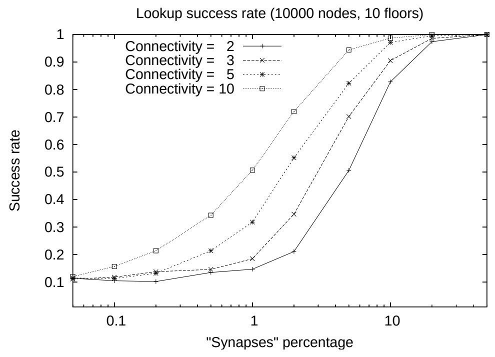


Figure 6. Exhaustiveness, $N = 10000$

5.7. iRho and Snake interpreters

Participants: Luigi Liquori [contact], Bernard Serpette.

iRho. We propose an imperative version of the Rewriting-calculus, a calculus based on pattern-matching, pattern-abstraction, and side-effects, which we call *iRho*. We formulate a static and a big-step "call-by-value" operational semantics of *iRho*. The operational semantics is deterministic, and immediately suggests how to build an interpreter for the calculus. The static semantics is given via a first-order type system based on a form of product-types, which can be assigned to terms like structures (*i.e.*, records). The calculus is *à la* Church, *i.e.* pattern-abstractions are decorated with the types of the free variables of the pattern. *iRho* is a good candidate for a core of a pattern-matching imperative language, where a (monomorphic) typed store can be safely manipulated and where fixed-points are built-in into the language itself. Properties such as determinism of the interpreter and subject-reduction are completely checked by a machine-assisted approach, using the Coq proof assistant. Progress and decidability of type-checking hold. The first interpreter for the imperative rewriting calculus was written in *Bigloo*; the metatheory was proved using the proof assistant *Coq*.
Snake. version 1.1 of *iRho*. It was completely rewritten in *Bigloo* by Luigi Liquori (new parser, new syntactic constructions, like, *e.g.*, guards, anti-patterns, anti-expressions, exceptions and parametrized pattern matching. *Snake* is a sugared version of *iRho* where all the keywords of the language are ASCII-symbols. It could be useful to teach basic algorithms and pattern-matching to children. See the web page http://www-sop.inria.fr/members/Luigi.Liquori/iRho/index.html.

6. New Results

6.1. Programmable overlay networks with ArchiNet

Participants: Didier Parigot, Baptiste Boussemart.

To provide an execution model of Overlay Network, it is necessary to extend the run-time of *SmartTools* (only Service Oriented Architecture of *SmartTools*). This new run-time, called *ArchiNet*, is underway of realization. This run-time corresponds to a very small part of *SmartTools*. *ArchiNet* is a natural extension of the SmartTools *SOA* to distributed execution on top of an overlay network. Specifically, on each machine, a JVM in our context, an instance of *ArchiNet* is created. But, the resource discovery (component) is performed through an overlay network. For the proper functioning of *ArchiNet*, each instance of Archinet on each machine is connected to an overlay network, for example *PiNet*. In each instance *ArchiNet*, available components (service) are published on the overlay network (*PiNet*). For all instances of *ArchiNet* into the Network, these components will be accessible via the discovery services. As the execution model of the *SOA SmartTools* is based on sending messages between components, *ArchiNet* lends naturally itself to a distributed execution. Given a set of discovery queries (discovery components or services), *ArchiNet* gives naturally in a execution support for these services (component) available on the network. As against the composition of these services remains the responsibility of the programmer. A paper describing the system is ongoing. See the web page http://www-sop.inria.fr/lognet/pon.

6.2. Strong typed communication with PiNet

Participants: Bernard Serpette, Didier Parigot.

In *PiNet* readers and writers on channels are described by abstract classes parametrized by the type of values transferred in the channel:

```
public abstract class InFlow<T> {
    public abstract T read();
}
public abstract class OutFlow<T> {
    public abstract void write(T value);
}
```

Readers and writers are certainly objects encapsulating transport specific values: a *socket* created by a *TCP* connexion for example. These values have a meaning only for the owned computer, they aren't *serializable*. Nevertheless, these readers and writers are built from simplest values (machine name, port number...) which are serializables. It is then possible to encapsulate these simple values in objects which will circulate in the network and will generate writers at the guessed place. In *PiNet* a writer generator is also described via an abstract class:

```
public abstract class OutFlowGenerator<T> implements Serializable {
    public abstract OutFlow<T> generate();
}
```

Here we mention explicitly, with the keyword `Serializable`, the fact that writer generators can be transferred in a network. The final stage consists in associating a reader to a writer generator of the same type:

```
public class EndPointFlow<T> {
    public InFlow<T> inFlow;
    public OutFlowGenerator<T> outGenerator;
}
```

This the key point of typing communication in *PiNet*: a reader and the associated writer generator are built by the same computer ensuring the soundness, *i.e.* readen values have the same types than the written one. This result has been published in [25].

6.3. DHT formal specification

Participants: Bernard Serpette, Cédric Tedeschi.

We have begun a formal specification associated to a *PiNet* implementation, of a *DHT* (Distributed Hash Table) oriented network. The specification is done via a relation on network states (small step semantics). A network state is the set of all individuals network nodes states. A node state consists of a, possibly sorted, set of neighbors, *i.e*. the nodes known by a specific node, a local memory and a queue of messages received by the node to be executed. The semantics describe the behavior of incoming messages in each nodes. The specificity of the formalized *DHT* are:

- as in Chord, the path found between any two nodes of the network has a length which is in $\mathcal{O}(Log(n))$, where n is the number of nodes of the network;
- the graph induced by the neighbors is symmetric, *i.e*. if a node a can communicate with a node b, then b can communicate with a. This fact is generally ensured by the transport layer (*TCP*, *UDP*...) and thus the associated improvement (path lengths are divided by two) comes at a low price;
- the dynamic nature of such network, *i.e.*, the fact that nodes can join and leave the network, requires to readjust the neighbors of some nodes. This readjustment is called stabilization. In contrast to Chord, where stabilization is done via a periodic background process which is hard to tune, our stabilization is done during the routing of messages and thus 1) does not make any administrative charge when the network is sleeping 2) uses only few words in already existing packets and so has a very limited impact to the average network latency.

We plan to use *Coq*, as in a previous work [18], to prove some properties on this specification (non dead-lock, reachability, stabilization convergence...).

6.4. Social overlay networks

Participants: Luigi Liquori, Cédric Tedeschi, Francesco Bongiovanni.

A significant part of today's Internet traffic is generated by peer-to-peer (*P2P*) applications, used originally for file sharing, and more recently for real-time multimedia communications and live media streaming.

Distributed hash tables (*DHT*s) or "structured overlay networks" have gained momentum in the last few years as the breaking technology to implement scalable, robust and efficient Internet applications. *DHT*s provide a lookup service similar to a hash table: (key, value) pairs are stored in the DHT, and any participating node can efficiently retrieve the value associated with a given key. Responsibility for maintaining the mapping from names to values is distributed among the nodes, in such a way that a change in the set of participants causes a minimal amount of disruption. This allows *DHT*s to scale to extremely large numbers of nodes and to handle continual node arrivals, departures, and failures.

Chord [47] is one of the simplest protocols addressing key lookup in a distributed hash table: the only operation that Chord supports is that given a key it route onto a node which is supposed to host the entry (key,value). Chord adapts efficiently as nodes join and leave the system. Theoretical analysis and simulations showed that the Chord protocol scales up logarithmically with the number of nodes. In Chord, every node can join and leave the system without any peer negotiation, even though this feature can be implemented at the application layer. Chord uses consistent hashing in order to map keys and nodes' addresses, hosting the distributed table, to the same logical address space. All the peers knows a unique hash function, representing the only way to map physical addresses and keys to an single logical address space. Peers can join the Chord just by sending a message to any node belonging to the Chord overlay. No reputation mechanism is required to accept, reject, or reward peers that are more reliable or more virtuous than others. Merging two Chord rings together is a costly operation because of the induced message complexity and the substantial time the distributed finger tables needs to stabilize. Both rings have to know their relative hash functions and have to decide which ring will absorb the other one, the latter point being critical because of the politics and security reliance's. We propose to connect smaller Chord networks in an unstructured way via special nodes playing the role of *neural synapses*.

6.4.1. Features

Schematically, the main Babelchord's features are:

Routing over SW/HW-Barriers. Namely, the ability to route queries through different, unrelated, *DHT*s (possibly separated by firewalls) by "crossing floors". A peer "on the border" of a firewall can bridge two overlays (having two different hash functions) that were not meant to communicate with each other unless one wants to merge one floor into the other (operation with a complexity linear in the number of nodes). The possibility to implement strong or weak security requirements makes Babelchord suitable to be employed in Internet applications where software or social barriers are an important issue to deal with.

Social-based. Every peer has data structures recording peers and floors which are more "attractive" than others. An "hot" node is a node which is stable (alive) and which is responsible for managing a large number of (keys-values) in all hosted *DHT*s. An "hot" floor is a floor responsible of a high number of successful lookups. Following a personal "good deal" strategy, a peer can decide to invite an hot node on a given floor it belongs to, or to join an hot floor, or even create from scratch a new floor (and then invite some hot nodes), or accept/decline an invitation to join an hot floor. This social-behavior makes the Babelchord network topology to change dynamically. As observed in other *P2P* protocols, like *Bittorrent*, peers with similar characteristics are more willing to group together on a private floor and thus will eventually improve their overall communications quality. Finally, the "good deal" strategy is geared up to be further enhanced with a reputation-system for nodes and floors.

Neural-inspired. Since every floor has a proper hash function, a Babelchord network can be thought as a sort of *meta overlay network* or *meta-DHT*, where inter floors connections take place via crossroad nodes, a sort of neural synapses, without sharing a global knowledge of the hash functions and without a time consuming floor merging. The more synapses you have the higher the possibility of having successful routing is.

6.4.2. Suitable applications

Because of the above original features, the following are examples of applications for which Babelchord can provide a good groundwork (in addition, of course, to all genuine Chord-based applications, like cooperative mirroring, time-shared storage, distributed indexes and large-scale combinatorial search).

Anti Internet censorship applications. Internet censorship is the control or the suppression of the publishing or accessing of information on the Internet. Many applications and networks have been recently developed in order to bypass the censorship: among the many we recall Psiphon (http://psiphon.ca), Tor (http://www.torproject.org), and many others. Babelchord can support such applications by taking advantage of intra-floor routing in order to bypass software barriers.

Fully Distributed social-networks applications. Social-networks are emerging as one of the Web 2.0 applications. Famous social networks, such as Facebook or LinkedIn are based on a client-server architecture; very often those sites are down for maintenance. Babelchord could represent a scalable and reliable alternative to decentralize key search and data storage.

Large-scale brain model and simulations. (Via a distributed, neural-based, network.) As well explained by R.D. DeGroot (Project founded by KNAW, Netherlands), supercomputers exist now with raw computational powers exceeding that of a human brain. Technological and production advances will soon place such computing power within the hands of cognitive and medical neuroscience research groups. For the first time it will be possible to execute brain-scale simulations of cognitive and pharmacological processes over millions and then billions of neurons - even at the biological model level. Babelchord could help modeling as a meta-overlay network the human brain. A paper has been written and submitted to an international workshop [24].

6.5. P2P resource discovery

Participants: Cédric Tedeschi, Eddy Caron [EPI GRAAL INRIA], Frédéric Desprez [EPI GRAAL INRIA], Franck Petit [EPI GRAAL INRIA].

The Distributed Lexicographic Placement (DLP)-Table is a *P2P* approach for the service discovery within large scale grids. It relies on a prefix tree structured overlay network. It provides load balancing, efficient mapping of nodes of the tree onto processors of the network and fault-tolerance mechanisms, formally proved to be self-stabilizing, *i.e.* converging to a correct topology in a finite time starting from an arbitrary topology and memory state. It has been initially developed within the INRIA GRAAL project team.

In collaboration with Eddy Caron, Frédéric Desprez and Franck Petit of GRAAL, we have written a chapter to appear in the future book entitled "Handbook of Research on P2P and Grid Systems for Service-Oriented Computing: Models, Methodologies and Applications" and published by IGI Global [23]. This chapter gives a more *popularizing* view of the system and its features.

6.6. SmartTools into Eclipse

Participants: Didier Parigot, Baptiste Boussemart.

All *SmartTools* features have been integrated into the *Eclipse* environment. This work demonstrates:

- The *SmartTools* features are not offered in the *Eclipse* environment (automatic generation of components with small DSLs). There is a perfect complementarity between *Eclipse* and *SmartTools*.
- *Eclipse* is not built above or as an *SOA*. There is a strong coupling between plugins and there is no use sending messages between plugins.
- The integration has been greatly facilitated by the *SOA* architecture of SmartTools. This shows that the *SOA* concept allows this type of integration with a low coupling between the two entities.
- *SmartTools* was only slightly modified to make this integration possible.
- Until here, the *SOA* and *DSL* concepts were not used much in the *Eclipse* development itself.

Finally, this integration provides a novel and innovative proof of concept and a new paradigm on his way for adoption by the community. This result has been presented at *Eclipse* conference 2008 [21].

6.7. Ariwheels: an overlay architecture for vehicular networks

Keywords: *Mobile ad hoc networks*, *content-based networks*.

Participants: Luigi Liquori [contact for the *Ariwheels* simulator], Claudio Casetti [Politecnico di Torino, Italy], Diego Borsetti [Politecnico di Torino, Italy], Carla-Fabiana Chiasserini [Politecnico di Torino, Italy], Diego Malandrino [Politecnico di Torino, Italy, contact for the *Ariwheels* client].

Arigatoni on wheels (*Ariwheels* for short) is an overlay architecture designed for a vehicular network underlay environment. *Ariwheels* provides efficient, transparent advertising and retrieves resources carried by on-board and roadside nodes. Consider an urban area in which a Mobile Ad hoc Network (*MANET*) is deployed. Such *MANET* is populated by both mobile users, *e.g.* pedestrians with hand-held devices, cars equipped with browsing/computational capabilities, public-transportation vehicles and roadside infrastructures such as bus stops. All devices are supposed to have a wireless interface. Depending on their mobility, they may also be equipped with a wired interface. Such is the case of wireless Access Points (*AP*s), which are installed at a bus stop, in order to provide connectivity either to users waiting for a bus or to the bus itself (hence to its passengers). In such settings, devices carried by cars and pedestrians play the role of mobile agent computers; roadside infrastructures (*AP*s) and public transportation vehicles (buses, trams, cabs...) act as agent brokers, although some distinctive behaviors have to be introduced.

An agent broker in *Ariwheels* is logistically represented by a bus stop, and its colony is composed by mobile agent computers that have registered to it when they were within radio range of the *AP* installed at the bus stop. However, to take into account the high mobility of the scenario and enhance its performance in terms of load balancing and service response time, we introduce an additional, *Ariwheels*-specific entity, the *mobile agent broker* (*mAB*). This unit is a public transport vehicle equipped with a scaled-down broker-like wireless device. Every mobile agent broker is *associated* to (*i.e.*, it has the same identity of) a single agent broker. Such association exists at the overlay level and holds throughout its bus route. Clearly, at the underlay level, connectivity between the mobile agent broker and the associated agent broker may at times be severed.

The main aim of the mobile agent broker is to introduce the novel concept of *"colony–room"*: a small subset of mobile agents computers with a wireless connection to the mobile agent broker (pedestrian users on the bus, or pedestrian/vehicles around the bus or traveling along the same bus direction during a traffic jam...). In addition, thanks to its mobility, the mobile agent broker can collect registrations from mobile agent computers that were too far from the *AP* of the associated agent broker, and, therefore, might had never had the chance to register to it. The mobile agent broker operates in tight coordination with infrastructure devices, *i.e.*, with agent brokers, and acts, in effect, as a colony-room for one of them.

The mobile agent broker collects (un)registrations, service requests and service offers from the agent computers within the colony–room. When a wireless connection has been established between the mobile agent broker and a roadside *AP* (not necessarily corresponding to the associated agent broker), the data path to the associated agent broker is again available and an information exchange takes place resulting in the updating of each other's data. Specifically, the following actions occur. Firstly, the associated agent broker merges the mobile agent broker's routing table with the one it currently carries. Then, the associated agent broker handles the registration/discovery information and generates the appropriate responses. Finally, depending on the response time, the responses are returned to the mobile agent broker before it leaves the wireless *AP* coverage, or the next time it connects to an *AP*: this will normally happen at the next bus stop.

Wireless devices may either be user terminals (laptops, hand-helds, sensors...) or higher-level units providing connection to the entities within the *Ariwheels* architectures. *mAB*s will be able to provide, via the *RDP* protocol, identities of *AC*s in its local area offering (in a *P2P* fashion) various information specific to the area where the bus stop is located, such as movie listings of local theaters, or lunch menus of nearby restaurants, or traffic jams. Therefore, mobile devices carried by passengers on a bus act as *AC*s registered to a *mAB* (the *AP* on the bus), and may primarily exchange information among themselves. If information cannot be found among *AC*s in the subcolony, *SREQ*s are relayed to the *AB* of which the *mAB* is a subcolony. As a consequence, a *mAB* holds a subset of routing table entries that can be found on the *AB*. Mobile users may want to access the wealth of information available through the *AB*s or the *mAB*, by first subscribing to the colonies governed by *AB*s, and then by sending service requests to (mobile) agent brokers.

In the *Ariwheels* architecture,

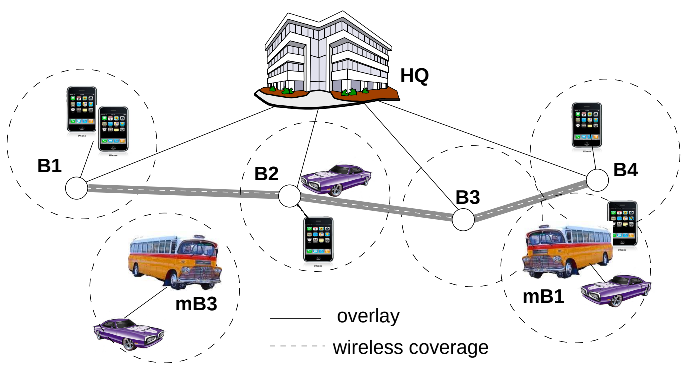


Figure 7.

A central coordination entity is located at a headquarter (*HQ*), in our case corresponding to the local transportation authority building. The coordination entity plays the role of a super-broker and it is provided with a wired connection to each of the 4 roadside *AP* at bus stops (tagged *B1* to *B4*). Mobile agent brokers (tagged *mB1* and *mB3*) shuttle between bus stops, each carrying a different broker association (tagged *B1* and *B3*), while mobile agents (portable devices or on-board car devices in the figure) are either connected to brokers or mobile agent brokers, depending on their mobility. This figure also shows the simulated city topology, that featured 6 bus stops with *AP*s, each corresponding to an agent broker. Furthermore, 3 buses acting as mobile agent brokers weave their own routes across the topology, among a population of as many as 60 vehicles acting as mobile agent computers. Each bus carries 10 passengers equipped with mobile agent computers capabilities, and it associates to the agent broker with the smallest colony at the time of departure from the bus station. The results have been published in [20], [22]. See the web page http://www-sop.inria.fr/members/Luigi.Liquori/ARIGATONI/Ariwheels.htm.

6.8. Powerful resource discovery for Arigatoni

Participants: Raphael Chand [Université de Geneva, Switzerland], Michel Cosnard, Luigi Liquori.

The version *V1* of the *RDP* protocol [2] enabled one service at the time to be requested, *e.g.* a *CPU*, or a specific file. In [4], the protocol was enhanced (*V2*) to take into account *multiple instances of the same service*. Adding multiple instances is a non trivial task because the broker must keep track (when routing requests) of how many resource instances were found in its own colony before delegating the rest of the instances to the surrounding colonies.

The version *V3* adds *multiple services* and *service conjunctions*. Adding service conjunctions allows a global computer to offer several services *at the same time*. Multiple services requests can be also asked to a *AB*; each service is processed sequentially and independently of others. As an example of multiple instances, a *GC* may ask for 3 *CPU*s, *or* 4 chunks of *1GB* of *RAM*, *or* one chunk of *10GB* of *HD*, *or* one gcc compiler; as an example of a service conjunction, a *GC* may ask for another *GC* offering *at the same time* one *CPU*s, *and* one chunk of *1GB* of *RAM*, *and* one chunk of *10GB* of *HD*, *and* one gcc compiler.

If a request succeeds, then via the orchestration language of *Arigatoni* (not described in this paper), the *AC* client can synchronize all resources offered by the servers *AC*'s. To sum up, we study

- A complete description of the resource discovery protocol *RDP V3*, which allows multiple instances, multiple services, and service conjunctions.
- A new version of the simulator taking into account the non trivial improvements in the resource discovery protocol.
- Simulation results that show that our enhanced protocol is scalable.

This result has been published in [14].

6.9. Conditional Logical Framework

Participants: Furio Honsell [Udine's Major, Italy], Marina Lenisa [Università di Udine, Italy], Luigi Liquori, Ivan Scagnetto [Università di Udine, Italy].

The Edinburgh Logical Framework LF of [40] was introduced both as a general theory of logics and as a metalanguage for a generic proof development environment. In this paper, we consider a variant of LF, called *Conditional Logical Framework* $\mathsf{LF}_{\mathcal{K}}$, which allows to deal uniformly with logics featuring *side-conditions* on the application of inference rules, such as *Modal Logics*. We study the theory of $\mathsf{LF}_{\mathcal{K}}$ and we provide proofs for subject reduction, confluence, and strong normalization. We illustrate how special instances of $\mathsf{LF}_{\mathcal{K}}$ allow for smooth encodings of Modal Logics both in Hilbert /Natural Deduction style.

The motivation for introducing $\mathsf{LF}_{\mathcal{K}}$ is that the type system of LF is too coarse as to the "side conditions" that it can enforce on the application of rules. Rules being encoded as functions from proofs to proofs and rule application simply encoded as lambda application, there are only roundabout ways to encode provisos, even as simple as that appearing in a *rule of proof*. Recall that a rule of proof can be applied only to premises which do not depend on any assumption, as opposed to a *rule of derivation* which can be applied everywhere. Also rules which appear in many natural deduction presentations of Modal and Program Logics are very problematic in standard LF. Many such systems feature rules which can be applied only to premises which depend solely on assumptions of a particular shape [36], or whose derivation has been carried out using only certain sequences of rules. In general, Modal, Program, Linear or Relevance Logics appear to be encodable in LF only encoding a very heavy machinery, which completely rules out any natural Curry-Howard paradigm, see *e.g.* [26]. As we will see for Modal Logics, LF_{K} allows for much simpler encodings of such rules, which open up promising generalizations of the proposition-as-types paradigm.

Apart from Modal Logics, we believe that our Conditional Logical Framework could also be very helpful in modeling dynamic and reactive systems: for example bio-inspired systems, where reactions of chemical processes take place only provided some extra structural or temporal conditions; or process algebras, where often no assumptions can be made about messages exchanged through the communication channels. Indeed, it could be the case that a redex, depending on the result of a communication, can remain stuck until a "good" message arrives from a given channel, firing in that case an appropriate reduction (this is a common situation in many protocols, where "bad" requests are ignored and "good ones" are served). Such dynamical behavior could be hardly captured by a rigid type discipline, where bad terms and hypotheses are ruled out *a priori*. A paper has been written and published [19], and the student Petar Maksimovic is starting a Ph.D. on extension of Logical Frameworks.

6.10. Adding trait inheritance to the Java language

Participants: Luigi Liquori, Arnaud Spiwack [Ecole Politechnique].

"Inside every large language is a small language struggling to get out ..." [41] ... " and inside every small language is a sharp extension looking for better expressivity ..."

In the context of *statically-typed, class-based languages* [42], we investigate classes that can be extended with *trait* composition. A trait is a collection of methods without state; it can be viewed as an *incomplete stateless class*. Traits can be composed in any order, but only make sense when imported by a class that provides state variables and additional methods to disambiguate conflicting names arising between the imported traits. We introduce *FeatherTrait Java* (*FTJ*), a conservative extension of the simple lightweight class-based calculus *Featherweight Java* (*FJ*) with *statically-typed traits*. In *FTJ*, classes can be built using traits as basic behavioral bricks; method conflicts between imported traits must be resolved *explicitly* by the user either by (i) aliasing or excluding method names in traits, or by (ii) overriding explicitly the conflicting methods in the class or in the trait itself. We also introduce *FeatherTrait Java with Interfaces* (*iFTJ*), where traits need to be type-checked only once, which is necessary for compiling them in isolation, and considering them as regular types, like *Java*-interfaces with a behavioral content.

FeatherTrait Java (*FTJ*) and *FeatherTrait Java with Interfaces* (*iFTJ*), conservatively extends the simple calculus of *Featherweight Java* (*FJ*) by Igarashi, Pierce, and Wadler with statically-typed traits. The main aim is to introduce the typed trait-based inheritance in a class-based calculus *à la Java*; the calculus features mutually recursive class declarations, object creation, field access, method invocation and override, method recursion through `this`, subtyping and simple casting. Just as with *FJ*, some of the features of *Java* that we *do not* model include assignment, interfaces, overloading, base types (`int`, `boolean`, `String`, etc.), `null` pointers, abstract method declaration, shadowing of superclass fields by subclass fields, access control (`public`, `private`, etc.), and exceptions. Since *FTJ* provides no operations with side effects, a method body always consists of `return` followed by an expression.

The main contributions of this research vein are

1. We define the calculus *FTJ*, a conservative extension of *FJ* featuring trait inheritance. Multiple traits can be inherited by one class, and conflicts between common methods defined in two or more inherited traits must be resolved *explicitly* by the user either by (i) aliasing or excluding method names in traits, or by (ii) overriding explicitly the conflicted methods in the class that imports those traits or in the trait itself.
2. We define a simple type system for *FTJ* that type-checks traits when imported in classes, resulting in a sharp and lightweight extension of the type system of *FJ*. This can be considered as a first step in adding a powerful but safe form of trait-based inheritance to the *Java* language.

3. We define the *FeatherTrait Java with Interfaces* calculus (*iFTJ*), a variation of *FTJ* and a conservative extension of *FJ*, which allows traits to be type-checked only once. Traits in *iFTJ* look like *Java*-interfaces with some partial behavior inside.
4. We define a type system for *iFTJ* that type-checks traits only once, in order to be compatible with compilation in isolation. In a nutshell, every trait is type-checked using a judgment which lists the signatures of methods that are *required* in order to *complete* the missing behavior of the trait itself.

An example of what traits can look like is

```
trait TA {String p( ){return this.r( )+this.s( )+this.q( );}

          String s( ){return ‘Java’;}  String r( )  String q( )}

trait TB {String r( ){return ‘Hallo World, my name is’;}

           String s( ){return ‘FeatherTrait Java’;};}
```

Traits `TA` and `TB` are type-checked only once, thus could be compiled in isolation; trait `TA` “defines” method `s`, and method `p` which “requires” methods `r` and `q` (declared as interfaces). They can be both imported in a class declaration as follows

```
class Presentation extends Object imports TA TB

 {;Presentation(){super();}

  String ciao( ){return this.p( );}

  String    s( ){return ‘FeatherTrait Java with Interfaces,’;}

  String    q( ){return ‘I hope you will like me’;}}
```

Multiple traits can be imported by one class, and conflicts between common methods, defined in two or more inherited traits, must be resolved *explicitly* by the user, either by aliasing or excluding method names in traits, or by overriding the conflicted methods in the class that imports those traits or in the trait itself. As such, the evaluation of `(new Presentation( )).ciao( )` will produce
“`Hallo World, my name is FeatherTrait Java with Interfaces, I hope you will like me`”.

The results have been published in [17], [16], and those papers are subject of study in the 21h CM course module called *Systèmes à objets* given by Luigi Liquori in winter 08 to the ENS Lyon students.

6.11. iRho: An Imperative Rewriting-calculus

Participants: Luigi Liquori, Bernard Serpette.

Although rewriting-based languages are less popular than object-oriented languages such as *Java*, *C#*, etc., for ordinary programming, they can serve as common typed intermediate languages for implementing compilers for rewriting-based, functional, object-oriented, logic, and other high-level modern languages.

Pattern-matching has been widely used in functional and logic programming (*ML*, *Haskell*, *Scheme*, *Curry*, or *Prolog*); it is generally considered to be a convenient mechanism for expressing complex requirements about the function’s argument, rather than a basis for an *ad hoc* paradigm of computation.

One of the main advantages of rewriting-based languages is *pattern-matching* which allows one to discriminate between alternatives. These languages permit non-determinism in the sense that they can represent a collection of results. That is, pattern-matching need not be exclusive, multiple branches can be “fired” simultaneously. An empty collection of results represents an application failure, a singleton represents a deterministic result, and a collection with more than one element represents a non-deterministic choice in the collection.

The *Rewriting-calculus* (*Rho*) [29], [30], [31], [32] integrates matching, rewriting, and functions in a uniform way; its abstraction mechanism is based on rewrite rule formation: in a term of the form $P \to A$, one abstracts over all the free variables of the pattern P (instead over a simple variable as in the Lambda-calculus). The Rewriting-calculus is a generalization of the Lambda-calculus since one may choose the pattern P to be a variable. If an abstraction $P \to A$ is applied to the term B, then the evaluation mechanism is based on (1) bind the free variables present in P to appropriate subterms of B to build a substitution θ, and (2) apply θ to A. Indeed, this binding is achieved by matching P against B. In rewriting-based languages, pattern-matching can be "customizable" with more sophisticated matching theories, *e.g.* building-in associativity and/or commutativity of equality.

The original *Rho* calculus is computationally complete, and, thanks to pattern-matching, Lambda-calculus and fixed-points can be encoded and type-checked by using *ad hoc* patterns. In fact, *Rho* is a direct generalization of the core of a typed (rewriting-based and functional) programming language (of the *ML*∪*Elan* family) in which, roughly speaking, an *ML*-like let becomes by default a let rec, by abstracting over a suitable pattern P; through pattern-matching, one can type-check many divergent terms (like Ω).

Table 1. Accessors and Destructors in Rho/Lambda Calculi

ops/form	Rho-calculus	Lambda-calculus
cons	$X \to Y \to (\mathsf{cons}\ X\,Y)$	$\lambda X.\,\lambda Y.\,\lambda Z.\,Z\,X\,Y$
car	$(\mathsf{cons}\ X\,Y) \to X$	$\lambda Z.\,Z\,(\lambda X.\,\lambda Y.\,X)$
cdr	$(\mathsf{cons}\ X\,Y) \to Y$	$\lambda Z.\,Z\,(\lambda X.\,\lambda Y.\,Y)$

One of the main features of the Rewriting-calculus is that it can deal with structuring and destructuring structures, like lists (we record only the names of the constructor and we discard those of the accessors). Since structures are built into the calculus, it follows that the encoding of constructor/accessors is simpler than the standard encoding in the Lambda-calculus. The Table 1 informally compares the untyped encoding of accessors in the two formalisms.

We presents the first version of the *Imperative Rewriting-calculus* (*iRho*), an extension of *Rho* with references, memory allocation, and assignment *à la ML*. To our knowledge, no similar study exists in the literature. The *iRho*-calculus is a powerful calculus, both at the syntactic and at the semantic level. It includes all the features of functional/rewriting-based languages with imperative aspects and pattern-matching facilities.

The controlled use of references, in the style of the *ML* language also gives the user the programming ease and expressiveness that might not a priori be expected from such a simple calculus.

The crucial ingredients of *iRho* are the combination of (i) modern and safe imperative features, which give full control over the internal data-structure representation, and of (ii) "matching power", which provides the main *Lisp*-like operations, like $\mathsf{cons}/\mathsf{car}/\mathsf{cdr}$. The language *iRho* provides a good theoretical foundation for an emerging family of languages combining rewriting, functions, and patterns with semi-structured *XML*-data or combining object-orientation and patterns with semi-structured data.

With this goal in mind we have *encoded* in *Coq* the static and dynamic semantics of *iRho*. All subtle aspects, which are usually "swept under the rug" on the paper, are here highlighted by the rigid discipline imposed by the Logical Framework of *Coq*. This process has often influenced the design of the semantics. The continuous interplay of mathematics and manual (*i.e.*, pen and paper) *vs.* mechanical proofs, and prototype implementations using high-level languages such as *Scheme* (and back) has been fruitful since the very beginning of our project. Although our calculus is rather simple, we expect to scale-up to larger projects, such as the certified implementation of compilers for a programming language of the *C* family.

Therefore, the main contributions of this research are:

- We provide a typed framework that enhances the functional language *Rho*, with imperative features like referencing, dereferencing, and assignment operators, and
- we enrich the type system with dereferencing-types and product-types. The resulting calculus *iRho* is a good candidate for giving a semantics to a family of functional, rewriting, and logic languages.

This result has been published in [15]. See the web page http://www-sop.inria.fr/members/Luigi.Liquori/iRho/index.html.

7. Other Grants and Activities

7.1. Regional Initiatives

7.1.1. PhD Grants PACA

Participant: Francesco Bongiovanni.

A *Bourses Doctorales cofinancées Région-Europe Organismes de recherche (BDO)* has been founded by the region PACA for the period 2008-2011. The thesis title is *Self-organizing overlay networks and generic overlay computing systems*.

7.2. National Initiatives

7.2.1. ANR blanc STAMP, 2007-2010

Participants: Didier Parigot, Ayoub Ait Lahcen [U. Rabat, Morocco and INRIA].

The overall objective of this ANR is to overcome present limitations of dynamic landscape modeling. The objective pursued by Didier Parigot is to explore new spatial and temporal primitives and the potential benefits that recent advances in Model-Driven Engineering can bring into the field of landscape studies. This should help to build concepts that will be formalized into domain-specific languages applicable to a wide range of landscape dynamics. See the web page http://tetis.teledetection.fr/index.php?option=com_content&task=view&id=421&Itemid=121.

A Ph.D. student will start his thesis on this topic on January 2009.

7.3. European Initiatives

7.3.1. Ph.D. Exchanges founded by Italian Research Founding Agency

Participant: Luigi Liquori.

Internazionalizzazione del Sistema Universitario, a common joint PhD project between the University of Udine, Siena, Pise (Italie), Valencia (Espagne), UNSA, Hyderabad (Inde). The project is founded by the Italian Ministry of Research and Education.

7.3.2. FP6 FET Global Computing: IST AEOLUS, 2005-2009

Participant: Luigi Liquori.

Algorithmic principles for building efficient overlay computers, in collaboration with 21 European universities and coordinated by University of Patras, Greece. LogNet participate to the package 2 (Resource management) and to the package 2 (Extending global computing to wireless users). See also LogNet highlights.

7.3.3. FP6 TEMPUS, 2007-2009

Participants: Luigi Liquori, Petar Maksimovic, Bojan Marinkovic [Math. Institute of Belgrade, Serbia].

Doctoral School Towards European Knowledge Society. Main aim of this Project, in collaboration with 6 European universities, is to promote the current European landscape of doctoral programmes in Serbia. Particularly, the Project will develop and implement a pilot Doctoral Programme according to the European innovative recommendations with comprehensive approach to information technologies, where foundational theories are fully integrated in a pragmatic engineering approach. LogNet is the head of the French chapter.

A Ph.D. student started his thesis on co-tutelle with the University of Novi Sad, and another Ph.D. will be visiting us during 2009. LogNet will be in charge to organize in 2009 the third training meeting in France.

7.4. Visitors

7.4.1. IN

- Giovanni Chiola, professor, U. Genova, 1 dd
- Marina Ribaudo, professor, U. Genova, 1 dd
- Matteo dell'Amico, Ph.D., U. Genova, now Post Doc Eurecom, 1 dd
- Mariangiola Dezani Ciancaglini, professor, U. Torino, 1 dd
- Luca Paolini, researcher, U. Torino, 1 dd

7.4.2. OUT

Luigi Liquori visited the following sites:

- Carnegie Mellon University CMU, Campus Qatar, 2 dd
- U. Roma 2, Italy, 1 dd
- CITI lab, Lyon, 2 dd
- LIG U. Grenoble, 2 dd
- INRIA Grenoble, 2dd
- INRIA Nancy Grand Est, 2 dd
- U. Paris 7, 5 dd
- U. Paris 12, 1 dd

8. Dissemination

8.1. Participation in committees

- Luigi Liquori is member of the *Commission de Spécialistes de la 27e section CNU* of the University of Nice Sophia Antipolis.
- Luigi Liquori is member of the *Commission de Spécialistes de la 27e section CNU* of the Ecole Nationale Mines de Nancy.
- Luigi Liquori is member of the *Commission de Spécialistes du jury CR2 INRIA Sophia Antipolis*.

8.2. Workshop organization

Luigi Liquori was co-chair of the Workshop *A journey through term rewriting and lambda-calculi*, Colloquium in Honor of Pierre Lescanne, see the web page http://www.loria.fr/equipes/cassis/PL60.

8.3. Participation in conference committees

- Luigi Liquori is PC member of the Sixth International Workshop on Hot Topics in Peer-to-Peer Systems HotP2P, 2009.
- Luigi Liquori is PC member of 3rd Conference on Algebra and Coalgebra in Computer Science, CALCO 2009.
- Didier Parigot is member of the 9éme Colloque Africain sur la Recherche en Informatique et en Mathématiques Appliquées, 2008.
- Cédric Tedeschi is PC member of the Eleventh International Symposium on Stabilization, Safety, and Security of Distributed Systems, SSS 2009.
- Cédric Tedeschi is PC member of the the International Conference on Computational Science, ICCS 2009.

8.4. Theses

The following theses are in preparation:

- Francesco Bongiovanni: Self-organizing overlay networks and generic overlay computing systems, since October 2008.
- Ayoub Ait Lahcen: Primitives spatiales, temporelles et multi échelles pour la modélisation des paysages dynamiques, since January 2009.
- Petar Macsimovic: Dealing with uncertain knowledge in Logical Framework, since December 2008.

8.5. Referees

- Luigi Liquori was a referee of the PhD thesis of Cédric Tedeschi.
- Luigi Liquori was a referee for the 10th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming.
- Luigi Liquori was a referee for the 19th International Conference on Rewriting Techniques and Applications.
- Luigi Liquori was a referee for the Quatriémes Journées Francophones de Programmation par Contraintes.
- Didier Parigot was a referee of the PhD thesis of Gautier Loyauté.

8.6. Teaching

- Luigi Liquori gave a course on *Systèmes à objets*, 21h CM, Master, Ens-Lyon.
- Luigi Liquori gave a 12,5 TD course on Peer to peer, Master 2 UNSA.
- Didier Parigot gave a 2h CM course on Service Oriented Architecture, Master 2 UNSA.

8.7. Invited talks

- Luigi Liquori presented *Toward Self-organizing Overlay Networks and Programmable Overlay Computing Systems* to the Carnegie Mellon University - Qatar (CMU-Q).

8.8. Participation in scientific meetings

- Luigi Liquori participated to the AEOLUS implementation meeting in Rome, Italy.
- Luigi Liquori participated to the AEOLUS preparation meeting in Athens, Greece.
- Luigi Liquori participated to the AEOLUS third year evaluation in Barcelone, Spain.

- Luigi Liquori participated to the DEUKS second training meeting in Valencia, Spain.

8.9. Participation in conferences

- Luigi Liquori presented *A Conditional Logical Framework*, to the International Conferences on Logic for Programming, Artificial Intelligence and Reasoning, LPAR 08, Doha, Qatar.
- Baptiste Boussemart and Didier Parigot presented the *SmartTools* Factory at Salon Linux, Paris.
- Didier Parigot presented the *SmartTools* Software Factory at EclipseCon 08, Santa-Clara, USA.

8.10. Spare presentations

- Luigi Liquori presented the LogNet team to the first INRIA-SAP Collaboration workshop.
- Luigi Liquori presented the LogNet team to a Chinese delegation (National University of Defense Technology, East China Normal University).
- Luigi Liquori and Didier Parigot presented the LogNet team to the ENS Lyon's students.
- Luigi Liquori and Didier Parigot presented the LogNet team to the Marseille Master's students.

9. Bibliography

Major publications by the team in recent years

[1] D. BENZA, M. COSNARD, L. LIQUORI, M. VESIN. *Arigatoni: Overlaying Internet via Low Level Network Protocols*, in "JVA, John Vincent Atanasoff International Symposium on Modern Computing", IEEE, 2006, p. 82–91.

[2] R. CHAND, M. COSNARD, L. LIQUORI. *Resource Discovery in the Arigatoni Overlay Network*, in "I2CS, International Workshop on Innovative Internet Community Systems", Lecture Notes in Computer Science, Springer-Verlag, 2006.

[3] R. CHAND, M. COSNARD, L. LIQUORI. *Powerful resource discovery for Arigatoni overlay network*, in "Future Generation Computer Systems", vol. 1, n^o 21, 2008, p. 31–38.

[4] R. CHAND, L. LIQUORI, M. COSNARD. *Improving Resource Discovery in the Arigatoni Overlay Network*, in "ARCS, International Conference on Architecture of Computing Systems", Lecture Notes in Computer Science, vol. 4415, Springer-Verlag, 2007, p. 98-111.

[5] M. COSNARD, L. LIQUORI, R. CHAND. *Virtual Organizations in Arigatoni*, in "DCM, International Workshop on Developpment in Computational Models. Electr. Notes Theor. Comput. Sci.", vol. 171, n^o 3, 2007.

[6] C. COURBIS, P. DEGENNE, A. FAU, D. PARIGOT. *Un modèle abstrait de composants adaptables*, in "TSI", vol. 23, n^o 2, 2004.

[7] C. COURBIS, D. PARIGOT. *La programmation générative pour le développement d'applications : les apports pour l'architecture*, in "ICSSEA", 2003.

[8] L. LIQUORI, D. BORSETTI, C. CASETTI, C.-F. CHIASSERINI. *An Overlay Architecture for Vehicular Networks*, in "IFIP Networking, International Conference on Networking", Lecture Notes in Computer Science, vol. 4982, Springer-Verlag, 2008, p. 60–71.

[9] L. LIQUORI, M. COSNARD. *Logical Networks: Towards Foundations for Programmable Overlay Networks and Overlay Computing Systems*, in "TGC, Trustworthy Global Computing", Lecture Notes in Computer Science, vol. 4912, Springer-Verlag, 2007, p. 90–107.

[10] L. LIQUORI, M. COSNARD. *Weaving Arigatoni with a Graph Topology*, in "ADVCOMP, International Conference on Advanced Engineering Computing and Applications in Sciences", IEEE Computer Society Press, 2007.

[11] P. NAIN, C. CASETTI, L. LIQUORI. *A Stochastic Model of an Arigatoni Overlay Computer*, Research Report, n^{o} to be given, Politecnico di Torino, 2007.

[12] D. PARIGOT, C. COURBIS. *Domain-Driven Development: the SmartTools Software Factory*, Technical report, n^{o} RR-5588, INRIA Sophia Antipolis, 2006.

[13] L. RIDEAU, B. P. SERPETTE, X. LEROY. *Tilting at windmills with Coq: Formal verification of a compilation algorithm for parallel moves*, in "Journal on Automated Reasoning", vol. 40, n^{o} 4, 2008, p. 307–326.

Year Publications

Articles in International Peer-Reviewed Journal

[14] R. CHAND, M. COSNARD, L. LIQUORI. *Powerful resource discovery for Arigatoni overlay network*, in "Future Generation Computer Systems", vol. 24, n^{o} 1, 2008, p. 31-38.

[15] L. LIQUORI, B. P. SERPETTE. *iRho: An Imperative Rewriting-calculus*, in "MSCS, Mathematical Structures in Computer Science", vol. 18, n^{o} 3, 2008, p. 467-500.

[16] L. LIQUORI, A. SPIWACK. *Extending FeatherTrait Java with Interfaces*, in "TCS, Theoretical Computer Science", Calculi, types and applications: Essays in honour of M. Coppo, M. Dezani-Ciancaglini and S. Ronchi Della Rocca, vol. 398, n^{o} 1-3, 2008, p. 243–260.

[17] L. LIQUORI, A. SPIWACK. *FeatherTrait: A Modest Extension of Featherweight Java*, in "TOPLAS, ACM Transaction on Programming Languages and Systems", vol. 30, n^{o} 2, 2008.

[18] L. RIDEAU, B. P. SERPETTE, X. LEROY. *Tilting at windmills with Coq: Formal verification of a compilation algorithm for parallel moves*, in "Journal on Automated Reasoning", vol. 40, n^{o} 4, 2008, p. 307–326.

International Peer-Reviewed Conference/Proceedings

[19] F. HONSELL, M. LENISA, L. LIQUORI, I. SCAGNETTO. *A Conditional Logical Framework*, in "LPAR, Logic for Programming, Artificial Intelligence, and Reasoning", Springer-Verlag, 2008, p. 143-157.

[20] L. LIQUORI, D. BORSETTI, C. CASETTI, C.-F. CHIASSERINI. *An Overlay Architecture for Vehicular Networks*, in "Networking", vol. 4982, Springer, 2008, p. 60-71.

Workshops without Proceedings

[21] D. PARIGOT. *SmartTools Software Factory*, in "EclipseCon'08, Short talk", 2008, http://www.eclipsecon.org/2008/?page=sub/&id=57.

Scientific Books (or Scientific Book chapters)

[22] D. BORSETTI, C. CASETTI, C.-F. CHIASSERINI, L. LIQUORI. *Content Discovery in Heterogeneous Mobile Networks*, in "Heterogeneous Wireless Access Networks: Architectures and Protocols", E. HOSSAIN (editor), Springer-Verlag, 2008.

[23] E. CARON, F. DESPREZ, F. PETIT, C. TEDESCHI. *DLPT: A P2P tool for Service Discovery in Grid Computing*, in "Handbook of Research on P2P and Grid Systems for Service-Oriented Computing: Models, Methodologies and Applications", N. ANTONOPOULOS, G. EXARCHAKOS, M. LI, A. LIOTTA (editors), To appear, IGI Global, 2009.

Other Publications

[24] L. LIQUORI, C. TEDESCHI, F. BONGIOVANNI. *Babelchord: a Social Tower of DHT-based Overlay Networks*, Submitted, 2008.

[25] D. PARIGOT, B. P. SERPETTE. *Qui séme la fonction, récolte le tuyau typé*, accepted at JFLA'09, 2008.

References in notes

[26] A. AVRON, F. HONSELL, M. MICULAN, C. PARAVANO. *Encoding Modal Logics in Logical Frameworks*, in "Studia Logica", vol. 60, n^{o} 1, 1998, p. 161-208.

[27] J. W. BACKUS. *The IBM 701 Speedcoding System*, in "J. ACM", vol. 1, n^{o} 1, 1954.

[28] D. BENZA, M. COSNARD, L. LIQUORI, M. VESIN. *Arigatoni: Overlaying Internet via Low Level Network Protocols*, Technical report, n^{o} 5805, INRIA, 2006.

[29] H. CIRSTEA, C. KIRCHNER, L. LIQUORI. *Matching Power*, in "RTA, International Conference on Rewriting Techniques and Applications", Lecture Notes in Computer Science, vol. 2051, Springer-Verlag, 2001, p. 77–92, http://www-sop.inria.fr/members/Luigi.Liquori/PAPERS/rta-01.ps.gz.

[30] H. CIRSTEA, C. KIRCHNER, L. LIQUORI. *The Rho Cube*, in "FoSSaCS, International Conference on Foundations of Software Science and Computation Structures", Lecture Notes in Computer Science, vol. 2030, Springer-Verlag, 2001, p. 168–183, http://www-sop.inria.fr/members/Luigi.Liquori/PAPERS/fossacs-01.ps.gz.

[31] H. CIRSTEA, C. KIRCHNER, L. LIQUORI. *Rewriting Calculus with(out) Types*, in "WRLA, International Workshop on Rewriting Logic and its Applications. Electr. Notes Theor. Comput. Sci.", vol. 71, 2002, http://www-sop.inria.fr/members/Luigi.Liquori/PAPERS/wrla-02.ps.gz.

[32] H. CIRSTEA, L. LIQUORI, B. WACK. *Rewriting Calculus with Fixpoints: Untyped and First-order Systems*, in "TYPES, International Workshop on Types for Proof and Programs", Lecture Notes in Computer Science, vol. 3085, Springer-Verlag, 2003, p. 147–161, http://www-sop.inria.fr/members/Luigi.Liquori/PAPERS/types-03.ps.gz.

[33] C. COURBIS, P. DEGENNE, A. FAU, D. PARIGOT. *Un modèle abstrait de composants adaptables*, in "TSI", vol. 23, n^{o} 2, 2004.

[34] C. COURBIS, P. DEGENNE, A. FAU, D. PARIGOT, J. VARIAMPARAMBIL. *Un modèle de composants pour l'atelier de développement SmartTools*, in "Systèmes à composants adaptables et extensibles", 2002.

[35] C. COURBIS, D. PARIGOT. *La programmation générative pour le développement d'applications : les apports pour l'architecture*, in "ICSSEA", 2003.

[36] M. CRESSWELL, G. HUGHES. *A companion to Modal Logic*, Methuen, 1984.

[37] H. V. DANG. *Transformation de l'Architecture Orientée Service de SmartTools (SOA) en des bundles OSGi*, Masters thesis, 2006.

[38] D. EPPSTEIN, Z. GALIL, G. ITALIANO. *Dynamic graph algorithms*, in "Handbook of Algorithms and Theory of Computation", chap. 22, CRC Press, 1998.

[39] M. FIORE, J. HÄRRI, F. FILALI, C. BONNET. *Vehicular Mobility Simulation for VANETs*, in "Annual Simulation Symposium", 2007, p. 301-309.

[40] R. HARPER, F. HONSELL, G. PLOTKIN. *A Framework for Defining Logics*, in "Journal of the ACM", Preliminary version in proc. of LICS'87, vol. 40, n^{o} 1, 1993, p. 143–184.

[41] A. IGARASHI, B. PIERCE, P. WADLER. *Featherweight Java: A Minimal Core Calculus for Java and GJ*, in "ACM TOPLAS", vol. 23, n^{o} 3, 2001, p. 396-450.

[42] L. LIQUORI. *Peter, le langage qui n'existe pas ... (Peter, the language that do not exists ...)*, Habilitation Thesis, INPL, 2007, http://www-sop.inria.fr/members/Luigi.Liquori/PAPERS/HDR-Liquori.pdf.

[43] M. OUAZARA. *Architecture-Orientée-Services Appliquée à la construction de RCPs réparties*, Masters thesis, 2007.

[44] D. PARIGOT, C. COURBIS. *Domain-Driven Development: the SmartTools Software Factory*, Technical report, n^{o} RR-5588, INRIA Sophia Antipolis, 2006.

[45] D. PARIGOT. *Software Factory on top of Eclipse: SmartTools*, in "Eclipse Technology eXchange workshop", may 2006.

[46] A. RAPOPORT. *Mathematical models of social interaction*, vol. II, John Wiley and Sons, 1963, p. 493–579.

[47] I. STOICA, R. MORRIS, D. KARGER, M. KAASHOEK, H. BALAKRISHNAN. *Chord: A Scalable Peer-to-Peer Lookup service for Internet Applications.*, in "ACM SIGCOMM", 2001, p. 149-160.

[48] S. TEAM. *SmartTools: Eclipse Plugin Factory*, October 2007, http://www-sop.inria.fr/teams/lognet/SmartTools/eclipse/index.html.

[49] W. M. P. VAN DER AALST, A. H. M. TER HOFSTEDE. *YAWL: yet another workflow language*, in "Information System", vol. 30, n^{o} 4, 2005, p. 245-275.

[50] J. VON NEUMANN. *The Principles of Large-Scale Computing Machines*, in "IEEE Ann. Hist. Comput.", vol. 10, n^{o} 4, 1988, p. 243–256.