Resource Discovery in the Arigatoni Overlay Network

Raphael Chand   Luigi Liquori
Michel Cosnard

INRIA, France
[Raphael.Chand,Luigi.Liquori,Michel.Cosnard]@inria.fr

Abstract: Arigatoni is a lightweight Overlay Network for dynamic and generic *Resource Discovery*. Entities in Arigatoni are organized in *Colonies*. A colony is a simple virtual organization composed by exactly one leader, offering some broker-like services, and some set of *Individuals*. Individuals are subcolonies of individuals, or basic units called *Global Computers*. Global computers communicate by first registering to the colony and then by mutually asking and offering services. The leader, called *Global Broker*, has the job to analyze service requests/responses coming from its own colony or arriving from a surrounding colony, and to route requests/responses to other individuals. After this discovery phase, individuals get in touch with each others without any further intervention from the system, typically in a P2P fashion. Communications over the behavioral units of the overlay network are performed by a simple *Global Internet Protocol*. Arigatoni provides fully decentralized, asynchronous and scalable resource discovery, that can be used for various purposes from P2P applications to more sophisticated Grid applications.

The main focus of this paper is to present the resource discovery algorithm used in Arigatoni, that is reminiscent to some algorithms employed in the publish/subscribe paradigm. We show some simulations that show that resource discovery in Arigatoni is efficient and scalable.

1 Introduction

Motivations. The *Global Computing Communication Paradigm*, *i.e*. computation via a seamless, geographically distributed, open-ended network of bounded resources owned by agents acting with partial knowledge and no central coordination is one of the most interesting challenges for the next decade. Aggregating many global computers sharing similar or different resources leads to a *Virtual Organization*. Moreover, organizing many overlay computers, using, *e.g*. tree- or graph-based topology leads to an *Overlay Network*.

The main challenge in this new field of research is how single resources, offered by the global/overlay computers are discovered. The process is called *Resource Discovery*: it requires an *up-to-date* information about widely-distributed resources. This is a challenging problem for very large distributed systems particularly when taking into account the continuously changing state of resources offered by global/overlay computers and the possibility of tolerating intermittent participation and dynamically changing status and

availability.

The first presentation of the Arigatoni overlay network was given in [2]. Reciprocity and hierarchical organization of the virtual organization in *Colonies*, governed by a clear leader (called *Global Broker*) are the main achievements of Arigatoni. Global computers belong to only one colony; requests for services and resources located in the same/another colony traverse a broker-2-broker negotiation which security is guaranteed via standard PKI mechanisms. Once the resource offered by a global computer has been found, the real resource exchange is performed out of the Arigatoni itself, *e.g.* in a P2P fashion.

In this paper, we explain how Arigatoni offers decentralized, asynchronous, and generic *resource discovery*. Once a global computer has issued a request for some services, Arigatoni finds some individuals that can offer the resources needed, and communicates their identities to the (client) global computer as soon as they are found.

The fact that Arigatoni only deals with resource discovery has one important advantage: the complete generality and independence of any given requested resource. Therefore, Arigatoni can fit with various scenarios in the global computing arena, from classical P2P applications, like file sharing, 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 GCU, the latter scenario being useful in case of catastrophic scenarios, like fire, terrorist attack, earthquake etc.

Arigatoni extends the pub/sub paradigm for resource discovery. Arigatoni takes inspiration by the *Publish/Subscribe* paradigm [9]; several pub/sub have been developed recently, such as XNet [7, 6], Siena [3] or IBM Gryphon [1]. In [10], the authors propose to adapt the Siena publish/subscribe system to achieve Gnutella-like resource discovery, by publishing queries to the notification service. In contrast, Arigatoni implements its own resource discovery algorithm, especially designed for generic and scalable resource lookup.

In Arigatoni, resource discovery works by asynchronously disseminating request messages in the system until some individuals have been found. More precisely, when global computers log in the system (a colony), they declare the list of services that they can offer. When a global computer asks for some services, it issues a service request message to its leader, without addressing it to any particular receiver. The system disseminates the message according to the services included in it *and* according to the services that the other global computers have declared. As a consequence, the communication model underlying Arigatoni extends conservatively pub/sub. Indeed, in the pub/sub paradigm, consumers subscribe to the system (typically called the *Notification Service*) to specify the type of information that they are interested in receiving. Producers publish data to the system. The notification service disseminates the data to all (if possible) the consumers that are interested in receiving it, according to the *content* of the data *and* the interests declared by the consumers. In Arigatoni, global computers “subscribe” to the system by declaring the services that they offer to serve. The same global computers also “publish” data in the system when they issue service requests. Arigatoni disseminates the data according to the services included in the requests and the services that the other global computers have

declared.

The pub/sub like communication form used in Arigatoni for resource discovery has several advantages. First, it allows Arigatoni to realize a full decoupling, in time, space, and synchronization, between the global computers. Second, due to its asynchronous nature, Arigatoni is, potentially, more scalable and can work in “disconnected” mode (*e.g.*, for mobile users and wireless devices). Third, indirect addressing makes it possible for the infrastructure to implement reliability, load balancing, fault-tolerance, persistence, or transactional semantics. More practically, since Arigatoni has a tree-like topology, we can use the pub/sub subscription mechanisms described in existing tree-based pub/sub systems such as XNet [6, 7, 4] or Siena [3], for subscription management, *i.e.*, for the construction and the update of *consistent* routing tables in the system. In addition, we can use the reliability mechanisms described in [7] to allow Arigatoni to be fault-tolerant or to adapt to dynamic topology changes.

However, one major difference between Arigatoni and classic pub/sub systems lies in their *functionality*. Indeed, the classic pub/sub paradigm deals with the publication of messages whereas Arigatoni focuses on *pure* resource discovery. More precisely, classic pub/sub systems aim at disseminating published messages to *all* interested consumers. In contrast, in Arigatoni, when a service request is issued, the goal is to find one (or maybe some) individuals able to provide the services included in the request, but not *all* the potential individuals. As a consequence, a much smaller fraction of the system is traversed. Besides, the routing strategy of the colony leader consists in always trying to find potential reseources in its own colony first. If it fails, it then delegates the request to its leader. This strategy is reminiscent of the *dynamic method lookup* employed in all Object-Oriented languages, and increases *resource encapsulation* inside colonies, another concept strongly related to Object Orientation.

Another major difference lies in the nature of the published events in classic pub/sub systems and the nature of service requests in Arigatoni. Indeed, in classic pub/sub systems, subscriptions are constraints on the set of all possible events. In contrast, in Arigatoni, service requests are also expressed as constraints. This latter point will be explained in more details in Section 3.

2 System units

Two different kinds of units compose the Arigatoni system: *Global Computer Units* (GCU), and *Global Broker Units* (GBU).

• A GCU is the basic peer of the global computing paradigm. It is typically a small device, like a PDA, or a PC, connected via IP.

• A GBU is the basic unit devoted to register and unregister GCUs, to receive service queries from client GCUs, to contact potential servant GCUs, to negotiate with the latter the given services, to trust clients and servers and to send all the information necessary to allow the client GCU, and the servants GCUs to communicate. Every GCU can register to only one GBU, so that every GBU controls a colony of collaborating global computers.

Hence, communication intra-colony is initiated via only one GBU, while communication inter-colonies is initiated through a chain of GBU-2-GBU message exchanges. In both cases, when a client GCU receives an acknowledgment for a requested service (with trust certificate) from the proper GBU, then the client will enjoy the service directly from the servant(s) GCU, *i.e.* without a further mediation of the GBU itself.

• A *Colony* is a simple virtual organization composed by exactly one leader and some set (possibly empty) of individuals. Colonies are organized in a tree structure where the root of a colony is its *leader*. Individuals are global computers (think it as an *Amoeba*), or subcolonies (think it as a *Protozoa*). An individual can be a GCU or a GBU (representing the leader of a a subcolony). GCUs cannot have children in the hierarchy. As such, GBUs can have both GBUs and GCUs as their children. As such, a colony has *exactly* one leader GBU and has at least one individual (the GBU itself), and may contains individuals (GCU's, or colonies).

• A *Community* is a raw set of colonies and global computers (think it as a *soup* of colonies and GCU without a leader). Starting from a community, the Arigatoni protocol allows individuals to dynamically aggregate in colonies. This topic has been addressed and formalized in [8].

The possibility for individuals to log/delog from a colony, or the possibility for a colony's leader to delog some "lazy" individuals makes *de facto* the network topology *dynamic*. This dynamicity implies that if GBUs hold routing tables about the services provided by their colony, particular care must be taken to maintain consistency when individuals log/delog. Moreover, due to the fact that individuals are not *slaves* but global computers with their own proper activity, a service request may lead to run-time failures. This happens when an individual gets busy by a local request, or when it suddenly delogs from the colony during the routing of the service request, or worst, when it gets hardware failures.

3 Resource discovery

Let $\mathcal{R}$ be the set of all possible resources (maybe infinite). GCUs provide resources by registering services to the system. A service S is a constraint on the set of resources. Let $match(S) \in \mathcal{R}$ be the set of resources that satisfy S. A GCU X that registers S announces that it can provide the set of resources $match(S)$. A GCU Y that issues a service request for service S' is looking for a resource that satisfies constraint S', *i.e.*, a resource in $match(S')$. If $match(S) \cap match(S') \neq \emptyset$, then there exists a resource that satisfies both S and S', and X can provide a resource to Y. We say that S and S' *overlap* iff $match(S) \cap match(S') \neq \emptyset$. For example $S = [\mathsf{Type} = \mathsf{CPU}] \wedge [\mathsf{Time} < 10\mathsf{s}]$ and $S' = [\mathsf{Type} = \mathsf{CPU}] \wedge [\mathsf{Time} > 5\mathsf{s}]$ overlap, since any resource with attribute Time between 5s and 10s matches.

The principle of resource discovery in Arigatoni is as follows. When a GCU sends a request for a set of services $S_1 \cdots S_n$, it builds a "ServiceRequest" message containing the set of services and sends it to its leader GBU. The message is then recursively processed by the GBUs in the system so as to find some individuals able to serve the services included in the request. The main basic principle of the protocol is that every GBU that receives a request

Algorithm 1 The resource discovery algorithm in the Arigatoni GIP protocol

```
1:  case Message is
    SREQ :
2:      ReturnPath{Message.Id} ← Message.Sender
3:      SendList ← SelectPeers(Message.Services, search_mode)
4:      for each (P, Serv(P)) ∈ SendList do
5:          Send ServiceRequest(Serv(P)) to P
6:      end for
7:      for each S∈Message.Services such that ∄(P, Serv(P))∈SendList, S∈Serv(P) do
8:          Append S to RejectList
9:      end for
10:     Send ServiceResponse({},RejectList) to ReturnPath[Id]
11: SRESP :
12:     for each S∈Message.AcceptedServices do
13:         if (S was not already accepted ) ∨ (EXHAUSTIVE_REPLY is set) then
14:             Append S to AcceptList
15:         end if
16:     end for
17:     SendList ← SelectPeers(Message.RejectedServices, intra_Colony_mode)
18:     for each (P, Serv(P))∈SendList do
19:         Send ServiceRequest(Serv(P)) to P
20:     end for
21:     for each S∈Message.RejectedServices such that ∄(P, S(P))∈SendList, S∈Serv(P) do
22:         Append S to RejectList
23:     end for
24:     Send ServiceResponse(AcceptList,RejectList) to ReturnPath[Id]
25: end case
```

always searches its own colony first to find the potential individuals able to serve the services included in the request. If no individuals are found, then the request is delegated to its leader GBU, and the process proceeds recursively. In addition, if the GBUs maintain some information about the services provided by their children, then they can transform a received request into sub-requests, so as to only ask a given child for the services that it (or its colony) provides.

The process eventually leads to some GCUs receiving a request. When one such GCU receives a request for some services, it chooses the services that it accepts to serve and the ones that it refuses to serve. It then sends a "ServiceResponse" message containing the list of accepted services and the list of rejected services, and sends it to its leader GBU. The response messages are then propagated recursively in the system, following the reverse path.

The resource discovery algorithm is the core of the GIP protocol; it is described in pseudo-code in Algorithm 1 and explained as follows. We only focused on the case of GBUs. The resource discovery algorithm in the case of GCUs is similar and has been voluntarily omitted (see [2] for details). Indeed, the involvement of GCUs in the process of resource discovery is limited to directly replying to request messages. Arigatoni only deals with the discovery of resources, while the real resource exchange is done in a P2P fashion. Let GBU N receive a message from a neighbor.

Case of Service Request (SREQ). We first consider the case of request messages. A request message received by GBU N means that N is asked to find some individuals to

provide the services included in the request. For that purpose, N first maps the "*Id*" of the request included in the message to the sender of the message (line 2), so as to allow reply messages to follow the reverse path of the request.

Line 3: Various intra colony search modes. The leader N then calls function "SelectPeers", taking as input the list of services included in the request message, $Message.Services$ (line 3). SelectPeers returns a list of pairs $\{(\mathsf{P}, Serv(\mathsf{P}))\}$, called *SendList*, where the first element P of a pair is the *Id* of a neighbor, and the second element $Serv(\mathsf{P})$ is a list of services, subset of $Message.Services$, that contains the list of services to ask to neighbor P. The *search_mode* determines the way function SelectPeers determines the *SendList*. The *search_mode* depends itself on whether P maintains some information about the services provided by its colony, *i.e.* a routing table. Currently, the following search mode are allowed: *broadcast* and *selective*, where the latter is itself sub-divided into three sub-modes: *exhaustive*, *greedy random*, and *greedy ordered*. If P does not maintain a routing table, then it has no other choice than to ask all its children for all the services included in the message, *i.e.*, to *broadcast* the request message. We will refer to this search mode as the *broadcast* mode. Now if P maintains a routing table that indicates which child leads to a potential individual able to serve a given service, then P can *selectively* send *customized* requests to its children. More formally, P only asks a child for a service that *overlaps* a service that it advertised, *i.e.* there exists a resource that satisfies both the requested service and the advertised service. We will refer to this mode as the *selective* mode. Consequently, P can choose some children and send them a request for the services that overlap the ones that they advertised. The selective search mode can then be refined as follows. Consider a particular service S included in the request message.

- In the *exhaustive* mode, P sends a request for service S to all the children that can serve it (*i.e.*, that contain potential individuals in their colony).
- In the *greedy random* mode, P sends a request for S to only one child that can serve the request, chosen uniformly at random.
- In the *greedy ordered* mode, P sends the request to only one child, chosen according to some predefined or *ad hoc* criteria (*e.g.*, depending on network factors, or according to the quantity of services that were accepted by each child, *à la* tit-for-tat).

In addition, we can refine even more the greedy modes, by introducing a parameter n, that defines the number of children to whom the request is sent. We could then define the *n-greedy random* or the *n-greedy ordered* modes. It is important to mention that the *SendList* variable can contain N's leader, call it L. That is, it may contain a pair $(\mathsf{L}, Serv(\mathsf{L}))$. For a particular service $S \in Serv(\mathsf{L})$, this happen when *no* child advertised some services that overlap S, *i.e.*, there are no potential individuals able to serve service S in N's colony. GBU N then *delegates* service S to its leader GBU. To prevent routing loops, the sender of the request message is never considered as a service provider.

Lines 4−6: Forwarding service request messages. Consequently, for each pair $(\mathsf{P}, Serv(\mathsf{P}))$ in the *SendList*, N sends to neighbor P a service request message for services $Serv(\mathsf{P})$ (lines $4 - 6$).

Lines $7 - 9$: Services rejection. Finally, each service S included in the request message, and such that no potential individual was found amongst N's neighbors, is reported as rejected by N, to the original issuer of the request message (lines $7 - 10$). Since N may

only maintain information about its own colony (apart the *Id* of the leader), this may only happen if N is the root of the topology or if the request message originated from N's leader.

Case of Service Response (SRESP). We now consider the case of reply messages. As previously explained, the process of propagating SREQ messages eventually leads to a certain number of GCUs receiving a request. Each GCU sends a reply message to its leader, with the list of accepted and the list of rejected services, along with its *Id*. Consequently, a given GBU N that participated in the propagation of the SREQ message eventually receives a certain number of SRESP messages from each of its children that was sent an instance of the (maybe transformed) SREQ message. Consider now an SRESP message sent to GBU N by a neighbor Q.

Lines $12-16$ **and** 24**: Reporting accepted services.** For each accepted service S, there are two different possibilities: either Q is the first child that accepted to serve the service, or the service was already accepted by some child other than Q. In the first case, N sends the reply back to the original sender or the request, reporting that service S has been accepted (lines 14 and 29). Otherwise, some neighbor other than Q already accepted to serve service S (*i.e.*, an individual in its colony). Then, if the EXHAUSTIVE_REPLY parameter flag is set (either in the GBU or included in the original request message), N also reports the reply back. Consequently, in the EXHAUSTIVE_REPLY mode, every GCU that accepted to serve a given service will be reported back to the GCU that issued the request. Otherwise, for each service asked in the request, only one single servant GCU will be reported. Furthermore, it is easy to add more flexibility by including a threshold $T_r{>}1$ on the number of replies. For example each GBU would report back T_r replies for the same service(s).

Lines 17**: Finding other individuals for rejected services.** We now consider the case of rejected services S. This means that in Q's colony, no potential individuals serving service S could be found, or no individuals accepted to serve it. Then, N has to find other neighbors that might contain individuals for service S. Consequently, N calls again function SelectPeers, with the list of rejected services as input (line 17). The function works as previously explained, except that it does not consider the peers (including Q) that were already sent a particular service. Also, logically enough, the services that were previously accepted are ignored. Finally, the original sender of the request is not considered (*i.e.*, $ReturnPathId$). Note that in the case where the *exhaustive* search mode is used, then the list $SendList$ returned by function SelectPeers may only contain a single pair $(\mathsf{L}, Serv(\mathsf{L}))$ (L is N's leader). Indeed, in the *exhaustive* search mode, all possible children in N's colony have already been asked for all the services included in the request message, that they can serve. Hence, rejected services are directly delegated to the leader L, if possible (*i.e.* if the latter was not the original sender of the request). The variable *SendList* contains a list of pairs $(\mathsf{P}, Serv(\mathsf{P}))$, where neighbor P is an individual that can potentially serve the services in $Serv(\mathsf{P})$, *and* has not been sent a request for any of them yet.

Lines $18-20$**: Forwarding request messages for rejected services.** Consequently, for each pair $(\mathsf{P}, Serv(\mathsf{P}))$ included in *SendList*, N sends to neighbor P a service request message for services $Serv(\mathsf{P})$ (lines $18-20$).

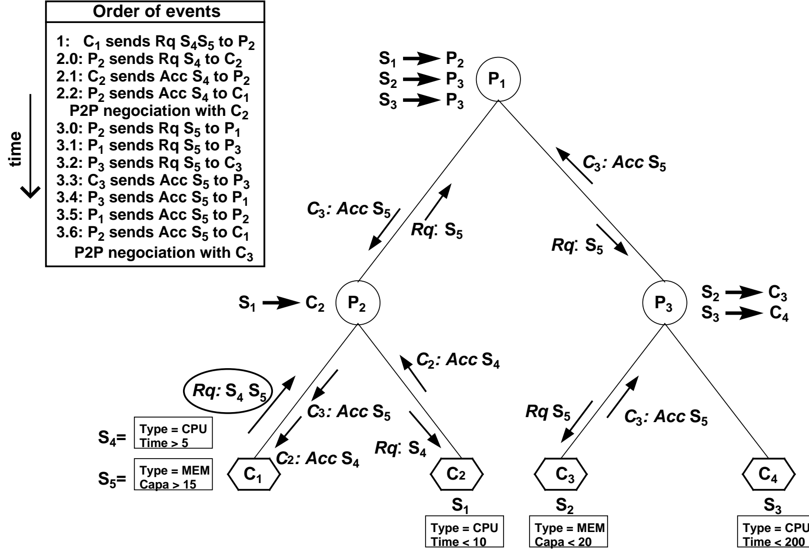


Figure 1: Resource discovery scenario.

Lines $21-23$ **and** 24**: Service rejection.** Finally, each service S included in the list of rejected services, and such that no additional potential individual could be found amongst N's neighbors, is reported as rejected by N, to the original issuer of the request message (lines $21-24$).

Example. Consider the example illustrated in Figure 1. Three GBUs are represented, namely $\mathsf{P}_1 \cdots \mathsf{P}_3$, and 4 GCUs, namely $\mathsf{C}_1 \cdots \mathsf{C}_4$. GCUs C_1 and C_2 (resp. C_3 and C_4) have P_2 (resp. P_3) as their leader, while P_1 is the leader of GBUs P_2 and P_3. GCUs C_2, C_3 and C_4 have registered services S_1, S_2 and S_3, respectively, and the routing tables of the upstream GBUs have been updated accordingly. In the example, resources are expressed as conjunctions of attribute/value pairs, and services are conjunctions of constraints on those attributes. We suppose that the *search mode* is set to *selective*, and we consider the scenario where GCU C_1 issues a service request for services S_4 and S_5, to its leader P_2. Since S_4 and S_1 overlap (any resource with 5<Time<10 satisfies both S_1 and S_4), GBU P_2 forwards a service request for service S_1 to GCU C_2. Note that given that S_5 and S_1 do not overlap, S_5 is not included in the request. Since P_2 does not find any GCU potentially able to serve S_5 (*i.e.*, no services in its routing table overlap with S_5), it delegates it to its leader GBU P_1. When C_2 accepts to serve S_4, it sends a reply message with its *Id* and the accepted service S_4, back to GBU P_2, which, in turn, forwards it back to C_1. Then C_1 can directly negotiate the resource with C_2. When P_1 receives the service request for S_5, it forwards it to P_3 (since S_2 and S_5 overlap), which in turn forwards it to GCU C_3. When C_3 accepts to serve S_5, the same process then repeats as for GCU C_2. Eventually, C_1

receives a reply message with the *Id* of GCU C_3 and the accepted service, namely S_5. We have an illustration of the asynchronous communication (C_1 received the reply messages independently of each others) and the encapsulation of resources in Arigatoni (GBU P_2 only searched for service S_4 in its own colony, *i.e.* GCU C_2).

Discussions, load balancing, scalability. We mainly focused on the resource discovery mechanism used in Arigatoni. Total decoupling between GCUs in space (GCUs do not know each others), time (GCUs do not participate in the interaction at the same time), and synchronization (GCUs can issue service requests and do something else, or may be doing something else when being asked for services) is a major feature of Arigatoni. Another important property is the encapsulation of resources in colonies. Those properties play a major role in the scalability of resource discovery in Arigatoni.

As stated before, the subscription mechanisms of classical tree-based pub/sub systems [6, 7, 4, 3] can be used for the maintenance and update of consistent routing tables. Furthermore, as for the reliability of subscription advertisement, we can adapt the reliability mechanisms described in [7] to allow Arigatoni to be fault-tolerant or to adapt to dynamic topology changes.

The reliability of the resource discovery mechanism itself, although desirable, is of lesser importance, given the fact that service provision is not guaranteed at all in Arigatoni. In other words, when a GCU issues a service request, it is possible that no individual is found for some of the services included in the request. This happens, for example, if those services were not declared by any GCUs in the system, or if all the GCUs that declared themselves as potential individual refuse to serve them. However, at the cost of memory and bandwidth requirements, it is still possible (future work) to implement reliable resource discovery by using a reliable transmission protocol (TCP), an acknowledgment scheme in combination with a retransmission buffer, and persistent data storage.

As defined above, GBUs are organized as a dynamic tree structure. Each GBU is a node of the tree, leader of its own subcolony and root of a subtree corresponding to the GBUs of its colony. It is then natural to address scalability issues that arise from that tree structure. In [5], we show that, under reasonable assumptions, the Arigatoni model is scalable. However, a complete performance evaluation is out of the scope of this paper and will rather be studied in a future work.

4 Protocol evaluation

To assess the effectiveness and the scalability of our resource discovery protocol, we have conducted simulations using large numbers of units and service requests.

Simulation setup. We have generated a network topology using the transit-stub model of the Georgia Tech Internetwork Topology Models package [11], on top of which we added the Arigatoni overlay network. The resulting network topology, contains 103 GBUs. GCUs

were not directly simulated in the network topology. Instead, to simulate the population of GCUs, we added a GCU *agent* to each GBU in the system. The GCU agent of a GBU represents the local colony of GCUs that are attached to that GBU as their leader.

We considered a finite set of resources $R_1 \cdots R_r$ of variable size r, and represented a service by a direct mapping to a resource. In other words, a service expresses the conditional presence of a single resource. We have a set of r services $\{S_1 \cdots S_r\}$, where service S_i expresses the conditional presence of resource R_i. A GCU declaring service S_i means that it can provide resource R_i. This simple model is still generic and sufficient for the main purpose of our experiments, which is to study the scalability of resource discovery in our system.

To simulate GCU load, we then randomly added each service with probability ρ at each GCU agent, and had it registered via the registration service of Arigatoni. The routing tables of the GBUs were updated starting at the initial GBU and ending at the root of the topology. In other words, it is as if each GBU has a probability ρ of having a GCU which registered service S_i, for any S_i. Thus, the parameter ρ can be seen as either the global availability of services, or as the density of population of GCUs (since the more the number of GCUs, the more likely it is that a given service is provided).

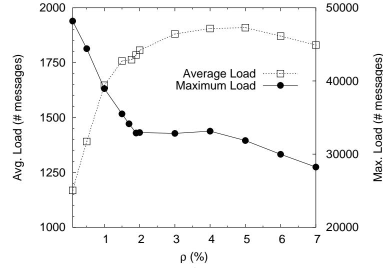
We then issued n service requests at GCU agents chosen uniformly at random. Each request contained one service (requests with k services can be seen as k service requests with one service), also chosen uniformly at random. Each service request was then handled by the resource discovery mechanism of Arigatoni described in Section 3. We used a service acceptation probability of $\alpha = 75\%$, which corresponds to the probability that a GCU that receives a service request *and* that declared itself as a potential individual for that service (*i.e.* that registered it), accepts to serve it.

The resource discovery algorithm was implemented in C++ and compiled using GNU C++ version 2.95.3. Experiments were conducted on a 3.0 Ghz Intel Pentium machine with 2 GB of main memory running Linux 2.4.28. The different experimental parameters are summarized in Figure 2. Upon completion of the n requests, we measured for each GBU its load as the number of requests (messages) that it received. We then computed the average load as the average value over the population of GBUs in the system. We also computed the maximum load as the maximum value of the load over all the GBUs in the system. Similarly, we computed the average and maximum load fractions as the average and max loads divided by the number of requests. The average load represents the average load of a GBU due to the completion of the n requests. The average load fraction represents the fraction of requests that a GBU served, in average. The maximum fraction represents the maximum fraction of the requests that a GBU served. Note that since a GBU receives at most one request message corresponding to a given service request, the average load fraction can be seen as the fraction of GBUs in the system involved in a service request, in average.

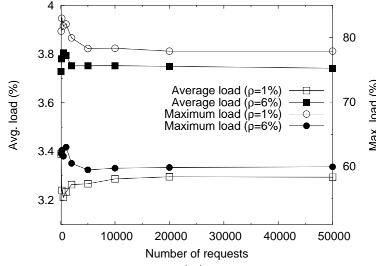
Finally, we computed the average service acceptation ratio as follows. For each GCU agent, we computed the local acceptation ratio as the number of service requests that yielded a positive response (*i.e.* the system found at least one individual), over the number of service requests issued at that GCU agent. We then computed the average acceptation

Vars	Description	Value
K	Number of GBUs	103
r	Size of services pool	128
ρ	Service availability	0.1% to 7%
α	Service accept. prob.	75%
n	Number of SREQ issued	100 to 50000

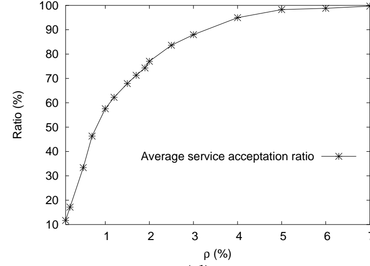
(a)



(b)



(c)



(d)

Figure 2: (a) Parameters of the experiments. (b) Average and maximum load w.r.t. service availability ρ. (c) Average and maximum load fraction w.r.t. the number of requests issued. (d) Average service acceptation ratio w.r.t. service availability ρ.

ratio as the average value over the number of GCU agents (that issued at least one service request).

We repeated the experiments for different values of ρ and n. Results are illustrated in Figure 2. Figure 2(b) and (d) were obtained with a fixed value of n of 50000 service requests.

Results and interpretations. Figure 2(b) shows the evolution of the average and maximum load when varying the service availability ρ. The maximum load was obtained for GBUs at the top of the leader hierarchy in the tree topology. It appears that the maximum load decreases with the service availability, while the average load increases. In other words, the load is more evenly distributed amongst the GBUs in the system. This is due to the strategy of our resource discovery mechanism which consists in always searching for individuals in its own colony first before delegating to its leader. Indeed, as the service availability increases, GBUs have a higher chance to find individuals in their own colony. Hence, the root leader (say GBU Top) of the topology participate less in the process of resource discovery, and the direct subleaders (say GBU SubTop) participate more. In other words, the resource discovery mechanism used in Arigatoni does not overload superleaders in the tree topology.

We also observe in Figure 2(b), for values of $2\%\leq\rho\leq4\%$, a *"plateau"* in the curve of the

maximum load, followed by a decreasing phase (ρ>4%), but with a much lower slope than before (ρ<2%). This is due to the fact that for ρ<2%, the root leader Top of the whole topology has the maximum load in the system. For ρ>2%, however, the immediate direct subleader Subtop takes over. This transition can be explained by the fact that for higher values of ρ, less messages are delegated to Top. At some point ($\rho \simeq 2\%$), the load of Top becomes less important than that of SubTop, due to the high number of colonies that the latter manages. The constantness observed in the curve around that value is probably due to the fact that a transition phase is necessary for SubTop to be sensitive again to the increase of ρ. The following decreasing period with a lower slope corresponds to the fact that SubTop is less sensitive to an increase of ρ (indeed, SubTop is mostly concerned with the availability of services in its own colonies).

Finally, we observe that the average load stabilizes, which shows that the system scales to large number of GCUs (since as previously mentioned, the service availability ρ can be assimilated to the number of GCUs in the system).

Figure 2(c) shows the average and maximum load fractions w.r.t. the number of service requests. It appears clearly that Arigatoni scales to large numbers of requests. In fact, the average number of requests received by a GBU increases linearly with the total number of requests, at a rate of $\sim 3.5\%$. In other words, in average, a GBU only receives $\sim 3.5\%$ of the total number of requests. Equivalently, only 3.5% of the overall population of GBUs in the system participate in the process of discovering a particular resource, in average. Figure 2(c) also shows that low level GBUs in the topology are not particularly overloaded (the most overloaded GBU manages 60% of the overall load for $\rho = 6\%$). Finally, it corroborates the assertion that higher values of ρ favor the maximum load over the average load, *i.e.*, load balancing gets more effective.

Figure 2(d) shows that, unsurprisingly, the average service acceptation ratio increases exponentially with the availability of services. This shows that Arigatoni is efficient in searching individuals for requested services. Indeed, a service availability of 4% enables the system to achieve an acceptation rate of 90%. In other words, the more the number of GCUs in the system, the more chances to find an individual for a service request.

5 Conclusion

In this paper, we presented the Arigatoni lightweight overlay network. We exposed in details the mechanisms that allow Arigatoni to offer dynamic and generic resource discovery. The main achievements are the complete decoupling between the different units in the system, and the encapsulation of resources in local colonies, which enable Arigatoni to be potentially scalable to very large and heterogeneous populations. We are currently improving Arigatoni with several new features, such as the possibility to ask a certain number of instances of a service (*i.e.*, the system should find the specified number of GCUs capable of providing that service), or the possibility to embed services in conjunctions (*i.e.*, the services in a conjunction should be provided by the same GCU). We are also working on the implementation of a real prototype and the subsequent deployment on the PlanetLab

experimental platform, and/or on GRID5000, the experimental platform available at the INRIA. As part of our ongoing research, we are also working on a more complete statistical study of our system, based on more elaborate statistical models and realistic assumptions.

Acknowledgment. The authors would like to thank Philippe Nain for its invaluable comments and interactions on the Arigatoni performance model. This work is supported by Aeolus FP6-2004-IST-FET Proactive.

References

[1] G. Banavar, T. Chandra, B. Mukherjee, J. Nagarajarao, R.E. Strom, and D.C. Sturman. An efficient multicast protocol for content-based publish-subscribe systems. In *Proc. of ICDCS*, 1999.

[2] D. Benza, M. Cosnard, L. Liquori, and M. Vesin. Arigatoni: A Simple Programmable Overlay Network. In *Proc. of John Vincent Atanasoff International Symposium on Modern Computing*. IEEE, 2006. To appear. Also as INRIA RR 5805.

[3] A. Carzaniga, D.S. Rosenblum, and A.L. Wolf. Design and Evaluation of a Wide-Area Event Notification Service. *ACM TOCS*, 19(3), 2001.

[4] R. Chand. *Large scale diffusion of information in Publish/Subscribe systems*. PhD thesis, University of Nice-Sophia Antipolis and Institut Eurecom, 2005.

[5] R. Chand, M. Cosnard, and L. Liquori. Resource Discovery in the Arigatoni Model. Technical Report 5924, INRIA, 2006.

[6] R. Chand and P. Felber. A scalable protocol for content-based routing in overlay networks. In *Proc. of NCA*, 2003.

[7] R. Chand and P. Felber. XNet: A Reliable Content-Based Publish/Subscribe System. In *SRDS 2004, 23rd Symposium on Reliable Distributed Systems*, 2004.

[8] M. Cosnard, L. Liquori, and R. Chand. Virtual Organizations in Arigatoni. *DCM: International Workshop on Developpment in Computational Models. Electr. Notes Theor. Comput. Sci.*, 2006. To appear.

[9] P. Th. Eugster, P. Felber, R. Guerraoui, and A.M. Kermarrec. The many faces of publish/subscribe. *Computing Survey*, 35(2):114–131, 2003.

[10] D. Heimbigner. Adapting publish/subscribe middleware to achieve gnutella-like functionality. In *SAC '01: Proc. of SAC*, pages 176–181, 2001.

[11] E.W. Zegura, K. Calvert, and S. Bhattacharjee. How to Model an Internetwork. In *Proc. of INFOCOM*, 1996.