Improving Resource Discovery in the Arigatoni Overlay Network

Raphaël Chand, Luigi Liquori, Michel Cosnard

Improving Resource Discovery in the Arigatoni Overlay Network*

Raphaël Chand[1]**, Luigi Liquori[2], and Michel Cosnard[2]

[1] University of Geneva, Switzerland,
Raphael.Chand@cui.unige.ch
[2] INRIA Sophia Antipolis, France,
[Michel.Cosnard,Luigi.Liquori]@inria.fr

Abstract. Arigatoni is a structured multi-layer overlay network providing various services with variable guarantees, and promoting an intermittent participation to the virtual organization where peers can appear, disappear and organize themselves dynamically. Arigatoni mainly concerns with how resources are declared and discovered in the overlay, allowing global computers to make a secure, PKI-based, use of global aggregated computational power, storage, information resources, etc. Arigatoni provides fully decentralized, asynchronous and scalable resource discovery, and provides mechanisms for dealing with dynamic virtual organizations. This paper introduces a non trivial improvement of the original resource discovery protocol by allowing to register and to ask for *multiple instances*. Simulations show that it is efficient and scalable.

1 Introduction

The explosive growth of the Internet gives rise to the possibility of designing large *overlay networks* and *virtual organizations* consisting of Internet-connected *global computers*, able to provide a rich functionality of services that makes use of its aggregated computational power, storage, information resources, etc. Arigatoni [3] is a structured multi-layer overlay network which provides resource discovery with variable guarantees in a virtual organization where peers can appear, disappear and organize themselves dynamically.

The virtual organization is structured in *colonies*, governed by *global brokers*, GB. A GB (un)registers global computers, GCs, receives service queries from clients GCs, contacts potential servants GCs, trusts clients and servers and allows the clients GC and the servants GCs to communicate. Registrations and requests are performed via a simple query language *à la* SQL and a simple *orchestration language à la* LINDA. Communication intra-colony is initiated via only one GB, while communication inter-colonies is initiated through a chain of GB-2-GB message exchanges. Once the resource offered by a global computer has

* This work is supported by Aeolus FP6-2004-IST-FET Proactive.
** This work was done while the author was at INRIA Sophia Antipolis, France.

been found in the overlay network, the real resource exchange is performed out of the overlay itself, in a peer-to-peer fashion.

The main challenges in Arigatoni lie in the management of an overlay network with a dynamic topology, the routing of queries and the discovery of resources in the overlay. In particular, resource discovery is a non trivial problem for large distributed systems featuring a discontinuous amount of resources offered by global computers and an intermittent participation in the overlay. Thus, Arigatoni features two protocols: the *virtual intermittence protocols*, VIP, and the *resource discovery protocol*, RDP. The VIP protocol deals with the *dynamic topology* of the overlay, by allowing individuals to login/logout to/from a colony. This implies that the process of routing may lead to some failures, because some individuals have logged out, or are temporarily unavailable, or because they have been *manu militari* logged out by the broker because of their poor performance or avidity (see [9]).

The total decoupling between GCs in *space* (GCs do not know each other), *time* (GCs do not participate in the interaction at the same time), and *synchronization* (GCs 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 overlay network. Another important property is the encapsulation of resources in colonies. All those properties play a major role in the scalability of Arigatoni's RDP.

The version V1 of the RDP protocol [6] enabled to ask for *one service* at the time, like, *e.g.* CPU or a particular file. The version V2, presented in this paper, allows *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 be found in the surrounding supercolonies.

As defined above, GBs are organized in a dynamic tree structure. Each GB, leader of its own subcolony, is a node of the overlay network, and a root of the subtree corresponding to its colony. It is then natural to address scalability issues that arise from that tree structure. In [6], we showed that, under reasonable assumptions, the Arigatoni overlay network is scalable. The technical contributions of this paper can be summarized as follows:

- A new version of the resource discovery protocol, called RDP V2, that allows for multiple instances; for example, a GC may ask for 3 CPUs, or 4 chunks of 1GB of RAM, or one compiler `gcc`. Multiple services requests can be also sent to a GB; each service will be processed sequentially and independently of others. If a request succeeds, then via the orchestration language of Arigatoni (not described in this paper), the GC client can synchronize all resources offered by the GC's servants.
- A new version of the simulator taking into account the non trivial improvements in the service discovery.
- Some simulation results that shows that our enhanced protocol is still scalable.

The rest of the paper is structured as follows: after Section 2 describing the main machinery underneath the new service request, Section 3 introduces the pseudocode of the protocol. Then, Section 4 shows our simulation results and Section 5 provides related work analysis and concluding remarks. An Appendix conclude with some auxiliary algorithms. For obvious lack of space, we refers to `http://www-sop.inria.fr/mascotte/Luigi.Liquori/ARIGATONI` for an extended version of this paper.

2 Resource Discovery Protocol RDP V2

Suppose a GC X registers to its GB and declares its availability to offer a service S, while another GC Y issues a request for a service S′. Then, the GB looks in its *routing table* and *filters* S′ against S. If there exists a solution to this matching equation, then X can provide a resource to Y. For example, $\mathsf{S} \triangleq [\mathsf{CPU{=}Intel}, \mathsf{Time{<}10sec}]$ and $\mathsf{S}' \triangleq [\mathsf{CPU{=}Intel}, \mathsf{Time{>}5sec}]$ match, with attribute values Intel and Time between 5 and 10 seconds. When a global computer asks for a service S, it also demands a certain number of instances of S. In RDP V2 this is denoted by "SREQ:$[(\mathsf{S}, n)]$".

Each GB maintains a table $\mathcal{T}$ representing the *services* that are registered in its colony. The table is updated according to the *dynamic registration and unregistration* of GC in the overlay. For a given S, the table has the form $\mathcal{T}[\mathsf{S}] = [(\mathsf{P}_j, m_j)]^{j=1\ldots k}$, where $(\mathsf{P}_j)^{j=1\ldots k}$ are the address of the direct children in the GB's colony, and $(m_j)^{j=1\ldots k}$ are the instances of S available at P_j.

For a service request SREQ:$[(\mathsf{S}, n)]$, the steps are:

- Look for q *distinct* GCs capable of serving S in the local colony.
- If $q{<}n$, then search r remaining instances $(n{-}q)$ in local subcolonies.
- If $r{<}(n{-}q)$, then delegate remaining instances $(n{-}q{-}r)$ to the leader of the colony.

A GC receiving a service request chooses the services that it *accepts/rejects* to serve. It then generates a SRESP message containing the lists of services accepted/rejected, and sends it to its GB. The response messages are then propagated back in the overlay, following the reverse path.

A Service Request SREQ:$[(\mathsf{S}, n)]$ may arrive bottom-up to the GB directly from its colony, or top-down from its own leader. In both cases, the leader tries to find n distinct GC that can serve S. More precisely, the list $[(\mathsf{P}_j, m_j)]^{j=1\ldots k}$ contains all the direct children in GB's colony that can serve S (child P_j with m_j instances of S). The discovery protocol features two search modes, *selective* and *exhaustive*. The selective mode is resource conservative at the price of important delays in case of low acceptance rates. The exhaustive mode is resource eager, but is independent of the acceptance rate. Let SREQ:$[(\mathsf{S}, n)]$, and $\mathcal{T}[\mathsf{S}] = [(\mathsf{P}_j, m_j)]^{j=1\ldots k}$. The selective mode consist in:

- If $\sum_{i=1}^{k} m_i \geq n$, then there are enough resources in the GB's colony to serve S. Let $y \leq k$ be the smallest index such that $\sum_{i=1}^{y} m_i \geq n$, and $\sum_{i=1}^{y-1} m_i < n$.

Then, SREQ:[(S, m_i)] is sent to all P_i with ($i \leq y-1$), and SREQ:[(S, $n - \sum_{i=1}^{y-1} m_i$)] is sent to P_y.
- If $\sum_{i=1}^{k} m_i < n$, then there are not enough GCs in the GB's colony that can serve S. Then, SREQ:[(S, m_i)] is sent to all P_i ($i \leq k$), and SREQ:(S, $n - \sum_{i=i}^{k} m_i$)] is delegated to the GB's leader. The rationale is that we first try to ask for *as many resources* in GB's colony, and then ask GB's leader for the *remaining resources*.

The exhaustive search mode consists in sending SREQ:[(S, $min(m_i, n)$)] to all P_i ($1 \leq i \leq k$), and to delegate SREQ:[(S, $n - \sum_{i=1}^{k} min(m_i, n)$] to the GB's leader. The rationale is to first ask for *all* resources in the GB's colony, and then ask the GB's leader for the remaining resources.

A Service Response SRESP:ACC:[(S, a)], or SRESP:REJ:[(S, d)], may follow service requests for services S. That is, "a" GCs accepted to serve S, and "d" denied. Due to the asynchrony of Arigatoni, more replies can arrive to the colony's leader (*i.e.* $a+d \geq n$). As for requests, there exists two modes that determine the way those acceptances are propagated back to the leader. In the *selective reply* mode, we return at most the number of instances of S that were asked by the leader whereas in the *exhaustive reply* mode, we return *all* acceptances.

As for acceptances there exists two modes that determine the way those acceptances are propagated back to the leader. In the *selective search* mode, the *whole colony* was asked for n instances of S, at most. This implies that exactly d instances of S must now be looked for to fulfill the original request. Hence, we first try to find d instances of S in other subcolonies. We then delegate the instances that could not be found to the leader. Finally, the remaining instances are reported back as rejected. In the *exhaustive search* mode, each *sub-colony* was asked for n instances of S, at most. Hence, there may be other sub-colonies that have not replied yet, and which may reply with enough acceptations to fulfill the request. The remaining instances must be delegated to the leader.

3 RDP pseudo-code

In this section, we detail the pseudo-code of the RDP V2. Five variables are used for each Arigatoni's interaction "ask-route-reply-route-back": *Path*, *asked*, *downstream*, *upstream*, and *SendList*. Each message (SREQ or SRESP) contains a unique identifier `id`, that is initially set by the GC that sends the initial SREQ message. Variable *Path* is a simple hash keyed by the identifier of the message. The other variables are double hashes which first key is the identifier of the message, and second key a given service S. The intuitive meaning of those variables is listed below.

- *Path*{`id`}: Peer address: identifies the child from which the original SREQ message came from.
- *asked*{`id`}{S}: Integer: number of instances of S asked and not replied.

Algorithm 1 Receiving $\mathsf{SREQ}_{\mathtt{id}}$:$[(\mathsf{S}, n)]$ from P_{from} (executed by P)

```
1:  Path{id} ← P_from                                   // To trace back the reverse route
2:  if SendList{id}{S} = ∅ then
3:      SendList{id}{S} ← Filter(S, P_from)             // Filter S in P's routing table
4:  end if
5:  (RoutingList, remaining) ← Route(P_from, S, n, search_mode)   // Build a routing list
6:  asked{id}{S} ← asked{id}{S} + n
7:  if remaining ≠ 0 then                               // Remaining instances to find
8:      if L ≠ ∅ and L ≠ P_from then                    // L exists and is different from P_from
9:          Insert L : (S, remaining) in RoutingList
10:         upstream{id}{S} ← upstream{id}{S} + remaining
11:     else                                            // P's colony is isolated
12:         Send SRESP_id:REJ:[(S, remaining)] to P_from
13:         asked{id}{S} ← asked{id}{S} − remaining
14:     end if
15: end if
16: for each Q : (S, m) ∈ RoutingList do
17:     Send SREQ_id:[(S, m)] to Q                      // Send SREQ_id to every element in RoutingList
18: end for
```

- *downstream*{id}{S}: Integer: instances of S asked in colony and not replied.
- *upstream*{id}{S}*:* Integer: instances of S delegated but not replied.
- *SendList*{id}{S}*:* (Peer address,Integer): the list of direct children that are potentially capable of serving S.

The pseudo-code of RDP V2 is showed in Algorithms $[1-5]$. For obvious lack of space, we details only Algorithms 1, and 2, and 3. The Appendix presents succintely the remaining auxiliary algorithms.

Case of service request (Alg. 1). Consider an individual P receiving a reply message $\mathsf{SREQ}_{\mathtt{id}}$ from a neighbor $\mathsf{P}_{\mathsf{from}}$, and let L be P's leader.

- In line 1, the originator of the request is first recorded in *Path*, so as to allow reply messages to follow the reverse path.
- In line 3, the *Filter* function (Alg. 4) determines the *SendList* corresponding to service S, *i.e.*, the list of direct children of P potentially able of serving S.
- In line 5, the *Route* function (Alg. 5) builds $(RoutingList, remaining)$, *i.e.*, the list of children that will be sent a particular service request, according to the selected search mode, and the positive number of the remaining instances for which no servant has been found. The *RoutingList* contains a list of mappings of the form Q:$[(\mathsf{S}, m)]$ which means that neighbor Q is to be sent a service request SREQ:$[(\mathsf{S}, m)]$.
- In line 8, if L exists and is not the originator of the request (to avoid routing loops), then the entry L:$(\mathsf{S}, remaining)$ is appended to the *RoutingList* (line 9), and the *upstream* counter is incremented accordingly (line 10); else (line 11, L exists and it is the originator of the request), since servants can be found for *remaining* instances of service S, a rejection reply is sent back to the originator of the request (line 12), and the *asked* counter is decremented accordingly (line 13).
- In line 17, a service request is sent to each neighbor Q having an entry in the *RoutingList*.

Algorithm 2 Receiving $\mathsf{SRESP}_{\mathtt{id}}$:ACC:$[(\mathsf{S}, a)]$ from $\mathsf{P}_{\mathsf{from}}$ (executed by P)

```
1:  case search_mode is
    "selective" :
2:      Send SRESP_id:ACC:[(S, a)] to Path{id}                    // Forward the SRESP
3:  "exhaustive" :
4:      if P_from = L then                                        // Top-down request
5:          upstream{id}{S} ← max(upstream{id}{S} − a; 0)
6:      else                                                      // Bottom-up request
7:          downstream{id}{S} ← max(downstream{id}{S} − a; 0)
8:      end if
9:      if asked{id}{S} ≥ a then                                  // More instances asked than accepted
10:         asked{id}{S} ← asked{id}{S} − a
11:         acc_return ← a
12:     else                                                      // More instances accepted than asked
13:         acc_return ← asked{id}{S} − a
14:         asked{id}{S} ← 0
15:     end if
16:     case reply_mode is
        "selective" :
17:         Send SRESP_id:ACC:(S, a) to Path{id}                  // Accepted "a" instances
18:     "hexaustive" :
19:         Send SRESP_id:ACC:(S, acc_return) to Path{id}         // Accepted "acc_return" instances
20:     end case
21: end case
```

Case of service response (Alg. 2,3). Consider an individual P receiving a reply message $\mathsf{SRESP}_{\mathtt{id}}$ from a neighbor $\mathsf{P}_{\mathsf{from}}$. The operation of the resource discovery algorithm is detailed in pseudo-code in Algorithms 2 and 3 and explained hereafter.

- *Acceptance (Alg. 2).* Let $\mathsf{SRESP}_{\mathtt{id}}$:ACC:$[(\mathsf{S}, a)]$ arrive from $\mathsf{P}_{\mathsf{from}}$ at P, *i.e.*, "a" global computers in P's colony accepted to serve service S.

  If the *selective search* mode was used to route the original service request $\mathsf{SREQ}_{\mathtt{id}} : (\mathsf{S}, n)$ (issued by $Path\{\mathtt{id}\}$), then the *whole colony* was asked for at most n instances of S. Hence, no more than n acceptances may arrive from P's colony. Thus, the reply message is simply forwarded back to $Path\{\mathtt{id}\}$ (line 2).

  If the *exhaustive search* mode was used, then *each child* was asked for at most n instances of S. Hence, it is possible that a number of acceptances higher than n arrives from L's colony. To do this, counters $asked, upstream, downstream$, and acc_return are updated accordingly (lines $5 - 14$).

  The *selective reply* mode simply reply back to $Path\{\mathtt{id}\}$ with "a" acceptation instances (line 17), while the *exhaustive reply* reply with "acc_return" instances (line 19).
- *Rejections (Alg. 3).* Let $\mathsf{SRESP}_{\mathtt{id}} : \mathsf{REJ}$:$[(\mathsf{S}, d)]$ arrive from $\mathsf{P}_{\mathsf{from}}$ at P, *i.e.*, "d" global computers in P's colony refused to serve S. This implies that *all* global computers in P's colony have been sent a request for S.

  If the sender of the message is the leader L, then no other potential servants for the d instances of S can be found. Consequently, the rejection message is simply forwarded back (line 2), and counters $asked$ and $upstream$ updated accordingly (lines 3 and 4).

Algorithm 3 Receiving $\mathsf{SRESP_{id}}$:REJ:$[(\mathsf{S}, d)]$ from $\mathsf{P_{from}}$ (executed by P)

```
1: if P_from = L then                                                    // Return rejections
2:    Send SRESP_id:REJ:[(S, d)] to Path{id}
3:    asked{id}{S} ← asked{id}{S} − d
4:    upstream{id}{S} ← upstream{id}{S} − d
5: else                                                  // Retry at other children or delegate
6:    case search_mode is
      "exhaustive" :                                                  // Try to delegate or reject
7:       downstream{id}{S} ← max(downstream{id}{S} − d; 0)
8:       if asked{id}{S} ≤ downstream{id}{S} + upstream{id}{S} then
9:                                                 // Less instances asked than down/upstream'ed
10:         Wait for more replies from other children
11:      else                                       // More instances asked than down/upstream'ed
12:         remaining ← asked{id}{S} − downstream{id}{S} − upstream{id}{S}
13:         if L ≠ ∅ and L ≠ Path{id} then
14:            upstream{id}{S} ← upstream{id}{S} + remaining
15:            Send SREQ_id:(S, remaining) to L
16:         else
17:            asked{id}{S} ← asked{id}{S} − remaining
18:            Send SRESP_id:REJ:(S, remaining) to Path{id}
19:         end if
20:      end if
21:      Remove P_from from SendList{id}{S}
22:   "selective" :                                          // Try other children, delete, or reject
23:      Remove P_from from SendList{id}{S}          // Don't send requests to P_from anymore
24:      (RoutingList, remaining) ← Route(P_from, S, d, search_mode)
25:      if remaining ≠ 0 then                          // Still some remaining instances to treat
26:         if L ≠ ∅ and L ≠ P_from then                  // L exists and is different from P_from
27:            Insert L : (S, remaining) in RoutingList
28:            upstream{id}{S} ← upstream{id}{S} + remaining
29:         else                                                          // P's colony is isolated
30:            Send SRESP_id:REJ:(S, remaining) to Path{id}
31:            asked{id}{S} ← asked{id}{S} − remaining
32:         end if
33:      end if
34:      for each Q : {(S, e)} ∈ RoutingList do
35:         Send SREQ_id:[(S, e)]to Q          // Send an SREQ for every element in RoutingList
36:      end for
37:   end case
38: end if
```

If L is not the sender of the rejected message, then there may be other potential servants in the colony or in other surrounding colonies. The operation of the protocol depends on the search mode that was used.

If the *exhaustive search* mode was used, then there are no other potential servants in L's colony but there may be some in other surrounding colonies. Hence, we first determine the number of instances of S that need to be found to fulfill the request.

If $asked \leq downstream + upstream$ (line 8), then there are enough potential servants in the colony or in surrounding colonies that have not replied yet, to fulfill the request. Consequently, we simply wait for more replies (line 10). In contrast, if $asked \geq downstream + upstream$, then we must look for more potential servants in order to fulfill the request. There are $(asked - downstream - upstream)$ of them to be found (line 12). As said before, those may only be found via a delegation to the leader L. Hence, the latter is sent a request for the remaining instances of S, if possible, (line 15), or a rejection

is sent back to the original sender of the request (line 18). The *upstream* or *asked* counters are updated accordingly (lines 14 and 17).
If the *selective* search mode is used, then there may be other potential servants in P's colony. The process is the same as in Algorithm 1, except that we do not consider children that have already been sent a request (line 21, 23). For that purpose, we use the *SendList* that was originally created by the *Filter* function (during the processing of the original service request message), and produce another *RoutingList* with the *Route* function (line 24).
Finally, we proceed as in Algorithm 1 (lines $25-36$).

4 Protocol Evaluation

The actual Arigatoni's topology is tree-based with routing complexity of $O(log\mathsf{N})$ (N being the number of nodes). However, in each GB, an extra complexity is due to solve the matching equation between the service request and the routing table $\mathcal{T}$ containing the mapping between peers and resources (this complexity is usually linear in the size of S).

To assess the effectiveness and the scalability of the protocol, we have conducted simulations using large numbers of units and service requests. For lack of space, we only present the results that correspond to the new features of the protocol, namely, the possibility to specify multiple instances of a service.

We have generated a network topology of 103 GBs, using the transit-stub model of the Georgia Tech Internetwork Topology Models package [18], on top of which we added the Arigatoni overlay network.

We take 120 distinct services, and we define the *overlap interval* $1 \leq \mathsf{L} \leq 120$, as the interval of indices inside which services match each other. That is, for all $(i,j) \in \mathsf{L}^2$, S_i and S_j match. If $\mathsf{L}=120$, then all services match each other; if $\mathsf{L}=1$, then each service only matches itself. At each GB, we added a random number of GCs chosen randomly between 0 and 100.

To simulate subscription load, we then randomly registered at each GC each service with a probability ρ denoting the *global availability of services*. We then randomly raised $50,000$ service requests per GC. Each request contained either a certain number of instances I of a service, chosen uniformly at random.

Each service request was then handled by the new RDP V2, described in this paper. We used a service acceptance probability of $\alpha=75\%$, which corresponds to the probability that a GC receiving a request for a service S (and offering S), accepts to serve it.

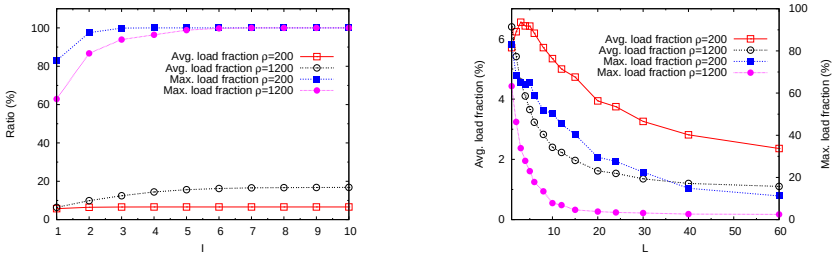
Upon completion of all the requests, we measured for each GB its load as the number of requests (messages) it received. We then computed the average load as the average value over the population of GB s in the system. We also computed the maximum load as the maximum value of the load over all the GBs in the system.

Similarly, we computed the average and maximum load fractions as the average and maximum loads divided by the number of requests. The average load represents the average load of a GB due to the completion of all the requests.

The average load fraction represents the fraction of requests that a GB served, on average. The maximum fraction represents the maximum fraction of the requests that a GB served.

We computed the average service acceptance ratio as follows. For each GC, we computed the local acceptance ratio as the number of service requests that yielded a positive response (*i.e.* the system found at least one GC), over the number of service requests issued at that GC agent. A service request that contained multiple instances of a service counts as a positive response only if the system found as many GCs as the number of instances specified in the request.

We then computed the average acceptance ratio as the average value over the number of GC (that issued at least one service request). We repeated the experiments for different values of ρ, L and I. Results are illustrated in Figures 1 and 2. The algorithm V2 was implemented in C++.

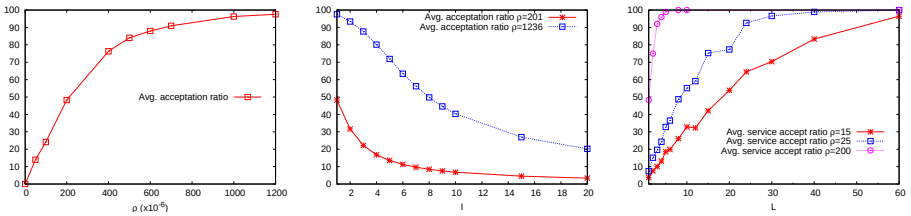


(a) (b)

Fig. 1. Average and maximum load fraction w.r.t. (a) number of instances of a service in service requests (b) overlap interval

Figure 1(a) shows the evolution of the average and maximum load fraction w.r.t. the number of instances of a service in service requests. Unsurprisingly, we observe that asking more instances of a service in a service request requires much more resource from the system. Indeed, for each instance, the system tries to find a different GC capable of providing the service. We observe that low-level GBs participate more, since there are more delegations. For values of I of circa 3 for ρ=0.02, and circa 6 for ρ=0.12%, the average and maximum load fractions stabilize. For values of I higher than those values, there are not enough resources in the system to completely fulfill the request (*i.e.*, not enough GCs capable of providing the requested service).

Figure 1(b) illustrates the evolution of the average and maximum load fractions w.r.t. the overlap interval L. Unsurprisingly, we observe that the more services match, the smaller the load imposed to the system. Indeed, for a given requested service, there are more potential candidates capable of providing a resource that satisfies it. For high enough values of L, the load stabilizes, and the resources are found very quickly (most often at the nearest GB).



(a) (b) (c)

Fig. 2. Average success rate (shown in the Y Axis in %) w.r.t. (a) service availability (b) number of instances (c) overlap interval

Figure 2(a) shows the average success rate w.r.t. the service availability ρ. Unsurprisingly, the average service acceptance 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 ρ=0.06% enables the system to achieve an acceptance rate of 90%.

Figure 2(b) shows the evolution of the success rate w.r.t. the number I of instances of S in a service requests. We observe that the average success rate decreases with the increasing of I and eventually stabilizes to 0%. This is due to the fact that the more instances we ask, the less GCs can be found to fulfill the request.

Figure 2(c) shows the average success rate w.r.t. the overlap interval L. We observe that the success rate increases with L, and eventually stabilizes to 100% at for higher values of L. This is due to the fact that the more services match, the higher the number of GCs capable of providing a resource that satisfies a given request.

5 Related Work and Conclusions

Many technologies, algorithms, and protocols have been proposed recently on resource discovery. Some of them focus on Grid or P2P oriented applications, but none of those targets the full generality of Arigatoni which only deals with generic resource discovery for building an overlay network of global computers, structured via a virtual organization of variable topology and clear distinct roles between leader and individuals (GCs or subcolonies).

Discussion on Closest Overlay Architectures (from [1]). The main challenges of pervasive computing are *how to build* an overlay network with respect to its topology, and *how to route queries* and *discover resources*.

There are essentially in the literature many basic types of overlays: structured (tree, ring or grid), unstructured, hybrid overlays (a combination of the two above), and multi-layer (or n-layer) overlays. Arigatoni falls in the latter category that is widely used in many P2P systems.

In a nutshell, in a *n-layer* overlay network, the responsibility assigned to Individuals differs (think of the different roles and responsibilities of GBs and GCs), since there are super-peers (GBs) serving as a server for a subset of all peers. Ordinary peers (GCs) submit queries to their super-peers and receive results from it. Super-peers are also connected to each others as ordinary peers (Individuals), routing messages over the overlay network, submitting, delegating, and answering queries on behalf of their peers (in their colony) and themselves. This structure is replicated *recursively*, creating a *n-layer topology*, where some peers become super-peers with decreasing responsibilities.

Typical issues of n-layer overlays are the size of each colony, together with the interests and the resources offered and demanded in each colony. Typical bottlenecks of n-layers are reliability and service availability (related to few points of failure) and load balancing. Classical solutions to cope with these problems are adding redundancy at the broker-layer. Historically, the n-layer topology generalizes the *two-layer topology*, such as the one we can find in the *hierarchical* DHT of Canon [12] and Coral [11].

Discussion on Closest Technologies. The Globus toolkit [13], is an open-source set of technology, protocols and middleware, used for building Grid systems and applications. Possible applications range from sharing computing power to distributed databases in a heterogeneous overlay network, where security is seriously taken into account. The toolkit includes stand-alone software for security, information infrastructure, resource management, data management, communication, fault detection, and portability.

The analogies with the Arigatoni model are in the *Community Scheduler Framework* component and the *Web Service Grid Resource Allocation and Management* of the toolkit concerning the resource discovery, and the *Globus Teleoperations Control Protocol* to allow units to cooperate (analogy with our *ad hoc* protocol). However, Globus does not target the full generality of Arigatoni, thanks to its generic, resource discovery algorithm that can be also suitable for pervasive compiting in addition to pure Grid-oriented applications.

Promoted by Sun, the JXTA [15] technology is a set of open peer-to-peer protocols that enable any device to communicate, collaborate and share resources. After a peer discovery process, any peer can interact *directly* with other peers. Hence, the overlay network of peers induced by the JXTA technology is *flat*.

Moreover, the main concern of the Arigatoni is the design of protocols for generic resource discovery, and intermittent participation, while the main concern of the JXTA technology is to offer some tools to implement a P2P model.

In addition, the Arigatoni focuses on the evolution/devolution of colonies and the mechanism of resource discovery, while JXTA technology allows peers to communicate using an already existing overlay network of peers. Arigatoni's aim is the dynamicity of the overlay network while JXTA's is the freedom of connectivity between peers. Finally, peers in the JXTA architecture come with their proper JXTA-ID (logical JXTA peers addressing) while Arigatoni relies on the more conventional IP addresses.

Pub/sub [10] is a communication paradigm for asynchronous dissemination of information. Consumers subscribe to the system (typically called the *Notification Service*) to specify the type of information that they are interested in. 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 data *and* the interests declared by the consumers.

Many pub/sub systems have been developed recently, such as XNet [8, 7], Siena [4] or IBM Gryphon [2]. In [14], the authors propose to adapt the Siena publish/subscribe system to achieve Gnutella-like resource discovery. Their work resembles ours in the sense that Arigatoni is also inspired by the pub/sub paradigm. However, in [14], resource discovery is achieved by publishing queries to the notification service. In contrast, Arigatoni implements its own resource discovery algorithm, especially designed for generic and scalable resource lookup.

Conclusions. In this paper, we describe the V2 of the Arigatoni's generic resource discovery protocol. The first version RDP V1 permitted to ask for *one service* at the time. The new improved protocol RDP V2 presented in this paper allows for *multiple instances*, the latter point being a non-trivial improvement. Other 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.

The reliability of the RDP V2 itself, although desirable, is of lesser importance, given the fact that service provision is not guaranteed at all in Arigatoni (indeed it is not a requirement). In other words, when a GC issues a service request, it is possible that no individuals are found for some of the services included in the request. This happens, for example, if those services have not been declared by any GCs in the system, or if all the GCs that have 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 (*e.g.* TCP), an applicative *acknowledgment scheme* in combination with a retransmission buffer, and persistent data storage, and leader's replication.

The subscription mechanisms of classical tree-based pub/sub systems [7, 5, 4] 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 [8] to allow Arigatoni to be fault-tolerant or to adapt to dynamic topology changes due to the intermittent participation of individuals [9].

We are currently still improving Arigatoni with several new features, such as the possibility to embed services in strong conjunctions (*i.e.*, the services in a strong conjunction should be provided by the *same* GC). We are also working on the implementation of a real prototype and the subsequent deployment on the PlanetLab experimental platform [16], and/or on GRID5000, the experimental platform available at the INRIA [17].

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, as well as the possibility to include some hierarchical DHT in addition to the routing tables. The possibility to change the Arigatoni topology from a hierarchical tree to a graph is also intriguing.

Acknowledgment. We would like to thank Luc Hohwiller for a careful reading of the paper, and the anonymous referees for the very useful comments.

References

1. AEOLUS. Deliverable D2.1.1: Resource Discovery: State of the art survey and Algorithmic Solutions, 2006. `http://aeolus.ceid.upatras.gr`.
2. 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.
3. 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*, pages 82–91. IEEE, 2006.
4. A. Carzaniga, D.S. Rosenblum, and A.L. Wolf. Design and Evaluation of a Wide-Area Event Notification Service. *ACM TOCS*, 19(3), 2001.
5. R. Chand. *Large scale diffusion of information in Publish/Subscribe systems*. PhD thesis, University of Nice-Sophia Antipolis and Institut Eurecom, 2005.
6. R. Chand, M. Cosnard, and L. Liquori. Resource Discovery in the Arigatoni Overlay Network. In *I2CS: International Workshop on Innovative Internet Community Systems*, volume LNCS. Springer, 2006. To appear. Also available as RR INRIA 5928.
7. R. Chand and P. Felber. A scalable protocol for content-based routing in overlay networks. In *Proc. of NCA*, 2003.
8. R. Chand and P. Felber. XNet: A Reliable Content-Based Publish/Subscribe System. In *SRDS 2004, 23rd Symposium on Reliable Distributed Systems*, 2004.
9. 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.
10. P. Th. Eugster, P. Felber, R. Guerraoui, and A.M. Kermarrec. The many faces of publish/subscribe. *Computing Survey*, 35(2):114–131, 2003.
11. M. J. Freedman and D. Mazières. Sloppy Hashing and Self-Organizing Clusters. In *Proc. of IPTPS*, pages 45–55, 2003.
12. P. Ganesan, P. Krishna, and H. Garcia-Molina. Canon in g major: Designing DHTS with Hierarchical Structure. In *Proc. of ICDCS*, 2004.
13. Globus Alliance. Globus Home Page. `http://www.globus.org/`.
14. D. Heimbigner. Adapting publish/subscribe middleware to achieve gnutella-like functionality. In *Proc. of SAC*, pages 176–181, 2001.
15. JXTA Community. JXTA Home Page. `http://www.jxta.org/`.
16. Planet Lab Consortium. Planet Lab Home Page, 2006. `http://www.planet-lab.org/`.
17. The Grid 5000 Consortium. Grid 5000 Home Page, 2006. `http://www.grid5000.org/`.
18. E.W. Zegura, K. Calvert, and S. Bhattacharjee. How to Model an Internetwork. In *Proc. of INFOCOM*, 1996.

A The *Filter* and *Route* Algorithms

Algorithm 4 *Filter*$(\mathsf{S}, \mathsf{P}_{\mathsf{from}})$

```
1: for each entry T[S'] = [(P_k, n_k)]^{k=1...C} in T do
2:    if S' matches S then
3:       for each j ≤ C such that P_j ≠ P_from do
4:          SendList{id}{S}{P_j} ← SendList{id}{S}{P_j} + n_j // (P_j, m) becomes (P_j, m + n_j)
5:       end for
6:    end if
7: end for
8: return SendList{id}{S}
```

Filter builds the *SendList* corresponding to service S (and interaction with identifier `id`), *i.e.*, the list of P's children that are potentially capable of serving S. The function parses all the services in the routing table accordingly.

Algorithm 5 *Route*$(\mathsf{P}_{\mathsf{from}}, \mathsf{S}, n, search\_mode)$

```
1: remaining ← n
2: RoutingList ← ∅
3: for each (Q, f) ∈ SendList{id}{S} do
4:    if Q = P_from or Q = Path{id} then
5:       continue                                   // Go to next iteration in loop
6:    end if
7:    case search_mode is
      "exhaustive" :
8:       if n ≥ f then                              // More instances asked than offered
9:          Insert Q : (S, f) in RoutingList
10:         remaining ← remaining − f
11:         downstream{id}{S} ← downstream{id}{S} + f
12:         Remove (Q, f) from SendList{id}{S}
13:      else                                       // More instances offered than asked
14:         Insert Q : (S, n) in RoutingList
15:         remaining ← 0
16:         downstream{id}{S} ← downstream{id}{S} + n
17:         f ← f − n
18:      end if
19:   "selective" :
20:      if remaining ≥ f then                      // More instances asked than offered
21:         Insert Q : (S, f) in RoutingList
22:         remaining ← remaining − f
23:         Remove (P, f) from SendList{id}{S}
24:      else                                       // More instances to offer than asked
25:         Insert Q : (S, remaining) in RoutingList
26:         f ← f − remaining
27:         remaining ← 0
28:      end if
29:      if remaining = 0 then                      // No more instances to treat
30:         break                                   // Break loop
31:      end if
32:   end case
33: end for
34: return (RoutingList, remaining)
```

Route builds *RoutingList*, *i.e.*, the list of neighbors that will be sent a particular service, according to the selected search mode; it has the form $\{(\mathsf{P}_i{:}(\mathsf{S}, n_i))\}^{i=1...h}$ that is neighbors P_i will receive a request for n_i instances of S. The function also returns the remaining instances for which no servant has been found.