Powerful Resource Discovery for Arigatoni Overlay Network

Raphael Chand, Michel Cosnard, Luigi Liquori

Powerful Resource Discovery for Arigatoni Overlay Network ⋆

Raphael Chand [a,1] and Michel Cosnard [b] and Luigi Liquori [b,*]

[a] *University of Geneva, Switzerland*

[b] *Inria Sophia Antipolis, France*

Abstract

Arigatoni is a structured multi-layer overlay network providing various services with variable guarantees, and promoting an intermittent participation in the overlay since peers can appear, disappear, and organize themselves dynamically. Arigatoni provides fully decentralized, asynchronous and scalable resource discovery; it also provides mechanisms for dealing with an overlay with a dynamic topology. This paper introduces a non trivial improvement of the resource discovery protocol by allowing the registration and request of *multiple instances of the same service*, *service conjunctions*, and *multiple services*. Adding multiple instances is a non trivial task since the discovery protocol must keep track (when routing requests) of peers that accept to serve and peers that deny the service. Adding service conjunctions allows a single peer to offer different services *at the same time*. Simulations show that it is efficient and scalable.

Key words: Overlay networks, Resource discovery, Virtual organizations, Dynamic graphs, Peer-to-peer, Global computing, Grid computing.

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

⋆ This work is supported by the AEOLUS FET Global Computing Proactive IST-015964, *Algorithmic Principles for Building Efficient Overlay Computers.*

* Corresponding author.

Email addresses: Raphael.Chand@cui.unige.ch (Raphael Chand), Michel.Cosnard@inria.fr (Michel Cosnard), Luigi.Liquori@inria.fr (Luigi Liquori).

[1] Work partly done while the author was at INRIA Sophia Antipolis, France.

that makes use of aggregated computational power, storage, information resources, etc. Arigatoni (1) 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. In a nutshell, the main units in Arigatoni are:

- A *Global Computer Unit,* GC, *i.e.* the basic peer of the global computing paradigm; it is typically a small device, like a PDA, a laptop or a PC, connected through IP in a various way (wired, wireless, etc.).
- A *Global Broker Unit,* GB, *i.e.* the basic unit devoted to subscribe and unsubscribe GCs, to receive service queries from client GCs, to contact potential server GCs, to negotiate with them services, to authenticate clients and servers, and to send all the information necessary to allow the client GC and the servers GCs to communicate. Every GB controls a *colony* of collaborating global computers. Hence, communication intra-colony is initiated via only one GB, while communication inter-colonies is initiated through a chain of GB-2-GB message exchanges whose security is guaranteed via PKI mechanisms. In both cases, when a client GC receives an acknowledgment of a service request from the direct leader GB, then the GC is served directly by the server(s) GC, *i.e.* without a further mediation of the GB, in a pure peer-to-peer fashion. Registrations and requests are performed via a simple query language *Ã la* SQL and a simple *orchestration language Ã la* LINDA, or BPEL.
- A *Global Router Unit,* GR *i.e.* the basic unit close to GCs and GBs that is devoted to send and receive packets, using the resource discovery protocol (2; 3), and to forward the "payload" to the units which are connected with this router. The connection GB-GR-GC is ensured via a suitable API.
- A *Colony* is a simple virtual organization composed of exactly one leader GB and a set (possibly empty) of individuals. Individuals are global computers (think it as an *Amoeba*), or sub-colonies (think it as a *Protozoa*). The two main characteristics of a colony are:

(1) a colony has *exactly* one leader GB and at least one individual (the GB itself);
(2) a colony contains individuals (GC's, or other sub-colonies).

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 intermittent 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 routing process may lead to 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 greediness (4).

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 (2) enabled one service at the time to be requested, *e.g.* a CPU, or a specific file. In (3), 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, presented in this paper, 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 GB; each service is processed sequentially and independently of others. As an example of multiple instances, a GC may ask for 3 CPUs, *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 CPUs, *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 GC client can synchronize all resources offered by the servers GC's. To sum up, the contributions of this paper are:

- 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.

The rest of the paper is structured as follows: after Section 2 describing the main machinery underneath the protocol features, Section 3 introduces the pseudocode of the protocol. Then, Section 4 shows our simulation results and finally Section 5 provides related work analysis and concluding remarks. This paper is an extended and improved version of (3).

2 Resource Discovery Protocol RDP V3

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 filter equation, then X can provide a resource to Y. For example, $\mathsf{S} \triangleq [\mathsf{CPU}{=}\mathsf{Intel}, \mathsf{Time}{<}\mathsf{10sec}]$ filters against $\mathsf{S}' \triangleq [\mathsf{CPU}{=}\mathsf{Intel}, \mathsf{Time}{>}\mathsf{5sec}]$, with attribute values Intel and Time between 5 and 10 seconds. In RDP V2, a global computer asks not only for a service S, but also for a certain number of instances of S; this is denoted by $\mathsf{SREQ}{:}[(\mathsf{S}, n)]$. In RDP V3:

- every GC registers in the colony with a *tuple* of *(services,instances)* like $\mathsf{SREG}{:}[(\mathsf{S}_i, n_i)]^{i=1\ldots h}$, and may ask for a tuple like $\mathsf{SREQ}{:}[(\mathsf{S}_j, n_j)]^{j=1\ldots k}$. Each service is processed sequentially and independently of others. This is achieved by wrapping the RDP V2 code inside a

$$\textbf{for each } j = 1 \ldots k \textbf{ do } \ldots \mathsf{V2}\, \textit{code} \ldots \textbf{ end foreach}$$

- a service request may also have the shape $\mathsf{SREQ}{:}[((\bigwedge_{i=1\ldots n} \mathsf{S}_i), n)]$, *i.e.* the system is no longer asked to find n occurrences of a single service, but rather n occurrences of a conjunction of services. That is, the system has to look for n distinct GCs, each GC being able to provide all the services in $\bigwedge_{i=1\ldots n} \mathsf{S}_i$.

Each GB maintains a *routing 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 single atomic service request $\mathsf{SREQ}{:}[(\mathsf{S}, n)]$, the steps are:

- Look for q *distinct* GCs able to provide S in the local GB's colony.
- If $q{<}n$, then search $r{\leq}(n{-}q)$ remaining instances in local sub-colonies.
- If $r{<}(n{-}q)$, then delegate $(n{-}q{-}r)$ remaining instances to the leader of the colony.

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

A Service Request $\mathsf{SREQ}{:}[(\mathsf{S}, n)]$ 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 locate n distinct GC that can provide S. More precisely, the list

$[(\mathsf{P}_j, m_j)]^{j=1...k}$ contains all the direct children in GB's colony that can provide S (child P_j with m_j instances of S).

The discovery protocol features two search modes, *selective* and *exhaustive*. Let SREQ:$[(\mathsf{S}, n)]$, and $\mathcal{T}[\mathsf{S}] = [(\mathsf{P}_j, m_j)]^{j=1...k}$.

- The selective search mode is resource conservative at the price of important delays in case of low acceptance rates. 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 provide 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:$[(\mathsf{S}, m_i)]$ is sent to all P_i ($i \leq y-1$), and SREQ:$[(\mathsf{S}, n - \sum_{i=1}^{y-1} m_i)]$ is sent to P_y.
  · If $\sum_{i=1}^{k} m_i < n$, then there are not enough GCs in the GB's colony to provide S. Then, SREQ:$[(\mathsf{S}, m_i)]$ is sent to all P_i ($i \leq k$), and SREQ:$[(\mathsf{S}, n - \sum_{i=i}^{k} m_i)]$ is delegated to the GB's leader. The rationale is that one first try to ask for *as many resources* in GB's colony, and then ask GB's leader for the *remaining resources*.
- The exhaustive mode is resource eager, but is independent of the acceptance rate. The exhaustive search mode consists in sending SREQ:$[(\mathsf{S}, min(m_i, n))]$ to all P_i ($1 \leq i \leq k$), and to delegate SREQ:$[(\mathsf{S}, n - \sum_{i=1}^{k} min(m_i, n))]$ 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:$[(\mathsf{S}, a)]$, or SRESP:REJ:$[(\mathsf{S}, d)]$, may follow service requests for services S. That is, "a" GCs accepted to provide 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 tell the way the acceptances are propagated back to the leader of the colony. In the *selective reply* mode, at most the number of instances of S that were asked by the leader are returned, whereas in the *exhaustive reply* mode, *all* acceptances are returned.

As for acceptances, there exists two modalities that determine the way those acceptances are propagated back to the colony's leader.

- In the *selective search* mode, the *whole colony* is 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, one first try to find d instances of S in other sub-colonies. One then delegate the instances that could not be found to the colony's leader. Finally, the remaining instances are reported back as rejected.
- In the *exhaustive search* mode, each *sub-colony* is 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 acceptances to fulfill the request. The

remaining instances must be delegated to the colony's leader.

3 RDP pseudo-code

In this section, we detail the pseudo-code of the RDP V3. Five global 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, which is initially set to the address of the GC that sends the initial SREQ message. The 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 is a given service S. The intuitive meaning of those variables is listed below.

- $Path\{\mathtt{id}\}$: *Peer address*: identifies the peer from which the original SREQ message came from.
- $asked\{\mathtt{id}\}\{\mathsf{S}\}$: *Integer*: instances of S asked and not replied, *i.e.*, the remaining number of instances of S to find to fulfill the request.
- $downstream\{\mathtt{id}\}\{\mathsf{S}\}$: *Integer*: instances of S asked in colony and not replied.
- $upstream\{\mathtt{id}\}\{\mathsf{S}\}$: *Integer*: instances of S delegated but not replied.
- $SendList\{\mathtt{id}\}\{\mathsf{S}\}$: $(Peer\ address, Integer)^*$: the list of direct children that are potentially able to provide S.

The pseudo-code of RDP V3 is showed in Algorithms $[1-8]$.

Algorithm 1 Receiving $\mathsf{SREQ}_{\mathtt{id}}{:}[(\mathsf{S}_i, n_i)]^{i=1\ldots k}$ from P_{from} (executed by P)

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

Case of service request (Algorithm 1). Consider a global broker P receiving a service request $\mathsf{SREQ}_{\mathtt{id}}$ from a neighbor $\mathsf{P}_{\mathsf{from}}$, and let L be P's leader.

The same steps are performed for each tuple $(\mathsf{S}, n) \in \mathsf{SREQ}$.

- In line 1, the originator of the request is first recorded in $Path\{\mathtt{id}\}$, so as to allow reply messages to follow the reverse path.
- In line 4, the *Filter* function (Algorithm 6) determines the $SendList\{\mathtt{id}\}\{\mathsf{S}\}$ corresponding to service S, *i.e.*, the list of direct children of the GB potentially able to provide S.
- In line 6, the *Route* function (Algorithm 8) builds $(RoutingList, remaining)$, *i.e.*, the list of children that will receive a particular service request, according to the selected search mode, and the positive number of the remaining instances for which no server has been found. The $RoutingList$ contains a list of mappings of the form Q:$[(\mathsf{S}, m)]$ which means that we send a service request SREQ:$[(\mathsf{S}, m)]$ to a neighbor Q.
- In line 9, 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 $RoutingList$ (line 10), and the *upstream* counter is incremented, accordingly (line 11); else (line 12, L exists and it is the originator of the request), since servers can be found for $remaining$ instances of service S, a rejection reply is sent back to the originator of the request (line 13), and the $asked$ counter is decremented, accordingly (line 14).
- In line 19, a service request is sent to each neighbor Q having an entry in the $RoutingList$.

Algorithm 2 Receiving $\mathsf{SRESP}_{\mathtt{id}}$:ACC:$[(\mathsf{S}_i, a_i)]^{i=1\ldots k}$ from $\mathsf{P}_{\mathsf{from}}$ (exec. by P)

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

Case of service response (Algorithms 2,3). Consider a global broker P receiving a reply message $\mathsf{SRESP}_{\mathtt{id}}$ from a neighbor $\mathsf{P}_{\mathsf{from}}$. The operation of

the resource discovery algorithm is explained hereafter. The same steps are performed for each tuple in SRESP.

- *Acceptance (Algorithm 2).* For each $(\mathsf{S}, a) \in \mathsf{SREQ}$, 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 provide S.

  If the *selective search* mode is used to route the original service request $\mathsf{SREQ}_{\mathtt{id}} : (\mathsf{S}, n)$, issued by $Path\{\mathtt{id}\}$, then the *whole colony* is 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 is used, then *each child* is 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 $6-15$).

  The *selective reply* mode simply replies back to $Path\{\mathtt{id}\}$ with a acceptance instances (line 18), while the *exhaustive reply* mode replies with *acc_return* instances (line 20).
- *Rejections (Algorithm 3).* For each $(\mathsf{S}, d) \in \mathsf{SREQ}$, let $\mathsf{SRESP}_{\mathtt{id}}$: REJ:$[(\mathsf{S}, d)]$ arrive from $\mathsf{P}_{\mathsf{from}}$ at P, *i.e.*, "d" global computers in GB P's colony refused to provide S. This implies that *all* global computers in P's colony have received a request for a service S.

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

  If L is not the sender of the rejected message, then there may be other potential servers in the colony or in other surrounding colonies. The operation of the protocol depends on the search mode that is used.
  - (*exhaustive search mode*) Then there are no other potential servers in L's colony but there may be in other surrounding colonies. Hence, the number of instances of S that need to be found to fulfill the request is first determined.

    If $asked \leq downstream+upstream$ (line 9), then there are enough potential servers in the colony or in surrounding colonies that have not replied yet, to fulfill the request. Consequently, we simply wait for more replies (line 11).

    In contrast, if $asked \geq downstream+upstream$, then one looks for more potential servers in order to fulfill the request. Then, there are $(asked - downstream - upstream)$ of them to be found (line 13). As said before, servers may be found by delegating to the leader L. Hence, the latter receives a request for the remaining instances of S, if possible, (line 16), or a rejection is sent back to the original sender of the request (line 19). The *upstream* or *asked* counters are updated, accordingly (lines 15 and 18).
  - (*selective search mode*) Then there may be other potential servers in P's

Algorithm 3 Receiving $\mathsf{SRESP_{id}}$:REJ:$[(\mathsf{S}_i, d_i)]^{i=1...k}$ from $\mathsf{P_{from}}$ (exec. 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:         for each (S, n) ∈ SRESP do
8:             downstream{id}{S} ← max(downstream{id}{S} − d; 0)
9:             if asked{id}{S} ≤ downstream{id}{S} + upstream{id}{S} then
10:                                                  // Less instances asked than down/upstream'ed
11:                Wait for more replies from other children
12:            else                                  // More instances asked than down/upstream'ed
13:                remaining ← asked{id}{S} − downstream{id}{S} − upstream{id}{S}
14:                if L ≠ ∅ and L ≠ Path{id} then
15:                    upstream{id}{S} ← upstream{id}{S} + remaining
16:                    Send SREQ_id:(S, remaining) to L
17:                else
18:                    asked{id}{S} ← asked{id}{S} − remaining
19:                    Send SRESP_id:REJ:(S, remaining) to Path{id}
20:                end if
21:            end if
22:            Remove P_from from SendList{id}{S}
23:        end for
24:    "selective" :                                  // Try other children, delete, or reject
25:        for each (S, n) ∈ SRESP do
26:            Remove P_from from SendList{id}{S}     // Don't send requests to P_from anymore
27:            (RoutingList, remaining) ← Route(P_from, S, d, search_mode)
28:            if remaining ≠ 0 then                  // Still remaining instances to treat
29:                if L ≠ ∅ and L ≠ P_from then       // L exists and is different from P_from
30:                    Insert L:(S, remaining) in RoutingList
31:                    upstream{id}{S} ← upstream{id}{S} + remaining
32:                else                                                  // P's colony is isolated
33:                    Send SRESP_id:REJ:(S, remaining) to Path{id}
34:                    asked{id}{S} ← asked{id}{S} − remaining
35:                end if
36:            end if
37:        end for
38:        for each Q:{(S, e)} ∈ RoutingList do
39:            Send SREQ_id:[(S, e)]to Q          // Send an SREQ for every element in RoutingList
40:        end for
41:    end case
42: end if
```

colony. The process is the same as in Algorithm 1, except that one do not consider children that have already received a request (line 22, 24). For that purpose, one use the *SendList* that is originally created by the *Filter* function (during the processing of the original service request message), and produce another *RoutingList* with the *Route* function (line 27).

Finally, one proceeds as in Algorithm 1 (lines $28 - 41$).

Algorithm 4 Receiving SREQ:$[(\mathsf{S}_i, 1)]$ from L (executed by a GC)

```
1: for each i = 1 . . . k do
2:     if accept then
3:         Acc ⟵^append S_i
4:     end if
5: end for
6: Send SRESP:ACC:[(S_i, 1)]^{i∈Acc} to L
7: Send SRESP:REJ:[(S_i, 1)]^{i∉Acc} to L
```

Algorithm 5 Receiving SRESP:ACC:$[(\mathsf{S}, a)]$ from L (executed by a GC)

1: Initiate P2P negotiation with GCs (embedded in message)

RDP **embedded in GCUs (Algorithms 4,5).** We show the cases of receiving a service request and a positive service response. The case of negative service response is trivial since the GC do simply nothing. Note that each reply message is formally of the form SRESP:ACC:$[(\mathsf{S}, \mathsf{P}_i)]^{i=1\ldots k}$ where the P_i are the GCs that accepted to provide S (the same for rejections). Those algorithms are quite intuitive and need not to be commented.

Algorithm 6 The *Filter*$(\mathsf{S}, \mathsf{P}_{\mathsf{from}})$ function for RDP V2

1: **for each** entry $\mathcal{T}[\mathsf{S}'] = [(\mathsf{P}_j, n_j)]^{j=1\ldots k}$ in $\mathcal{T}$ **do**
2: **if** S *filters* S' **then**
3: **for each** $j = 1 \ldots k$ such that $\mathsf{P}_j \neq \mathsf{P}_{\mathsf{from}}$ **do**
4: $SendList\{\mathtt{id}\}\{\mathsf{S}\}\{\mathsf{P}_j\} \leftarrow SendList\{\mathtt{id}\}\{\mathsf{S}\}\{\mathsf{P}_j\} + n_j$ // *Add/update* $SendList\{\mathtt{id}\}\{\mathsf{S}\}\{\mathsf{P}_j\}$
5: **end for**
6: **end if**
7: **end for**
8: **return** $SendList\{\mathtt{id}\}\{\mathsf{S}\}$

Algorithm 7 The *Filter*$(\mathsf{S} \triangleq (\bigwedge_{i=1\ldots n} \mathsf{S}_i), \mathsf{P}_{\mathsf{from}})$ function for RDP V3

1: **for each** $i = 1 \ldots n$ **do**
2: $tmp \leftarrow 0$ // *Auxiliary vector*
3: **for each** entry $\mathcal{T}[\mathsf{S}'] = [(\mathsf{P}_j, n_j)]^{j=1\ldots k}$ in $\mathcal{T}$ **do**
4: **if** S_i *filters* S' **then** // *Handle all conjunctions*
5: **for each** $j = 1 \ldots k$ such that $\mathsf{P}_j \neq \mathsf{P}_{\mathsf{from}}$ **do**
6: $tmp[j] \leftarrow tmp[j] + n_j$
7: **end for**
8: **end if**
9: **end for**
10: **if** $SendList\{\mathtt{id}\}\{\mathsf{S}\} = \emptyset$ **then**
11: $SendList\{\mathtt{id}\}\{\mathsf{S}\} \leftarrow tmp$
12: **else**
13: **for each** $j = 1 \ldots k$ **do**
14: $SendList\{\mathtt{id}\}\{\mathsf{S}\}\{\mathsf{P}_j\} = min(SendList\{\mathtt{id}\}\{\mathsf{S}\}\{\mathsf{P}_j\}, tmp[j])$
15: **end for**
16: **end if**
17: **end for**
18: **return** $SendList\{\mathtt{id}\}\{\mathsf{S}\}$

The *Filter* function for V2 builds the $SendList\{\mathtt{id}\}\{\mathsf{S}\}$ corresponding to the request `id` for a service S, *i.e.* the direct list of GB P's children that are potentially able to serve the request for S coming from $\mathsf{P}_{\mathsf{from}}$. The function parses all the services in the routing table, accordingly. The *Filter* function for V3 enables service conjunctions and for this it has to be modified. For a service request of the form SREQ:$[((\bigwedge_{i=1\ldots n} \mathsf{S}_i), n)]$, the system is no longer asked to find n occurrences of a single service, but rather n occurrences of a conjunction of services. That is, the system has to look for n distinct GCs, each GC being able to provide all the services in $\bigwedge_{i=1\ldots n} \mathsf{S}_i$. A conjunction of services is treated atomically, *i.e.*, as a single service S. Both algorithms are quite intuitive and they are are described in Algorithms 6 and 7.

Algorithm 8 *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)
```

The *Route* function of Figure 8 builds *RoutingList*, *i.e.*, the list of neighbors that ask for a particular service, according to the selected search mode; it has the form $\{(\mathsf{P}_i{:}(\mathsf{S}, n_i))\}^{i=1\ldots h}$, 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 server has been found.

4 Protocol Evaluation

The actual Arigatoni's topology is tree-based with a routing complexity of $O(log\,\mathsf{N})$ (N being the number of nodes). However, in each GB, an extra complexity is required in order to solve the filter 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 ability to specify multiple instances of a service, service conjunctions, and multiple services.

We have generated a network topology of 103 GBs, using the transit-stub model of the Georgia Tech Internetwork Topology Models package (5), on top of which we added the Arigatoni overlay network. We considered a finite set of services $\mathsf{S}_1 \ldots \mathsf{S}_r$ of size $r = 128$, with an exact filtering policy (*i.e.*, S_i filters S_i and no other services), and we defined the *overlap interval* $1 \leq \mathsf{L} \leq 128$, as the interval of indices inside which services filter each other, that is, for all $(i, j) \in \mathsf{L}^2$, S_i filters against S_j. If $\mathsf{L}=128$, then all services filter each other; if $\mathsf{L}=1$, then each service only filters with itself. At each GB, we added a number of GCs chosen randomly between 0 and 100.

At each GB, we added a random number of GCs chosen uniformly at random 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*, or as the density of population of GCs (since the more the number of GCs, the more likely it is that a given service is provided). The routing tables were updated, accordingly.

We then issued $50,000$ service requests at GCs chosen uniformly at random. Each request contained either a certain number of instances I of a service, or one instance of a conjunction of services, also chosen uniformly at random. Each service request is then handled by the RDP V3. We used a service acceptance probability of α=75%, which corresponds to the probability that a GC, receiving a request for a S (and offering S), accepts to provide 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.

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 the n 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. Since a GB receives at most one request message corresponding to a given service request, the average load fraction can be seen as the fraction of GBs in the system involved in a service request, in average.

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. A service request that contained

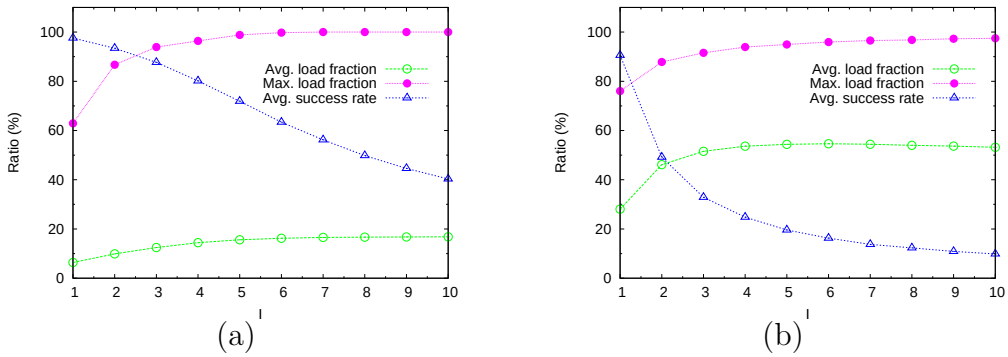

Fig. 1. Average and maximum load fraction, and average success rate w.r.t. (a) number of instances (b) number of services in conjunction.

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). Figure 1(a) shows the influence of the number of instances l in service requests on the average and maximum load fraction and on the average success rate. It is obtained with a value of ρ of 0.12%. Unsurprisingly, we observe that asking for more instances of a service requires more resources from the system. Indeed, for each instance, the system tries to find a different GC able to provide the service. We observe that low level GBs participate more, since there are more delegations. For values l higher than 7, the average and maximum load fractions stabilize, as the average success rate keeps decreasing; this means that there are not enough resources in the system to completely fulfill the request (*i.e.*, not enough GCs able to provide the requested service).

Figure 1(b) shows the influence of the number of services in a conjunction. It is obtained with a value of ρ of 3%. The phenomenon and its explanation is mostly similar to that of Figure 1(a), except that it happens at a much greater scale. Indeed, the system must find a GC that can provide (and accepts) all the services in a conjunctive service request, which requires to probe a much greater portion of the network than if a single service is asked.

5 Related Work and Conclusions

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

Discussion on Closest Overlay Architectures (from (6)). The main challenges of "pervasive computing" are *how to build* an overlay network with dynamic topology, and *how to route queries* and *discover resources* efficiently.

In an overlay network, any message is routed through the full overlay; as such, the topology adopted in the overlay strongly affects routing algorithms and their complexity. The overlay is built on top of the physical one, and, thus, two neighbor nodes in the overlay network may be many links apart in the physical network. The Arigatoni topology is a dynamic *hierarchical n-layer tree*. To assist lookup, structured overlays map (key of) data item to nodes (our GBs). Hence, the mapping is usually done through hashing the key space of the data item to the id space of nodes. In Arigatoni, routing tables denoting the set of resources are stored in GB's; thus, each GB maintains a partition of the data space. When a GC asks for a resource, the query is *filtered* against the first direct GB's routing table; in case of *filter-failure*, the query is recursively forwarded to the direct super-GB. Any answer of the query must follow the reverse path. Thus, 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 performance of lookup, caching and replication of either data, search paths, or both is possible. Besides improving routing, replication assists in providing load balancing, improves fault tolerance, and the durability of data items.

In the literature, there are essentially the following 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 between GBs and GCs), since 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; they route messages over the overlay network, submit, delegate, and answer queries on behalf of their peers. This structure is replicated *recursively*, creating a *n-layer topology*, where peers become super-peers with decreasing responsibilities.

Typical issues in n-layer overlays are the size of each colony, and the internal coherence of the resources offered and requested by each colony. Typical bottlenecks of n-layers are reliability, 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 most related tree topologies are BATON (7) and P-GRID (8), whereas the closest n-layer topologies are the one of Canon (9) and Coral (10).

We summarize the most closest topologies.

- (BATON) is a balanced binary tree that features a left and a right routing table, both contained in each node (denoted by a single logical id). Nodes may join or leave the network at any time, provided the tree remains balanced. The node receiving a join can forward the join towards a node which has less children or which is a leaf node. This implies that a GC can become a GB. Leaving the network is constrained to not breaking the balanced tree unless finding a substitute. As such, load balancing can be costly.
- (P-GRID) is a distributed dynamic binary search tree, such that the search space is partitioned between peers. The salient feature of P-GRID is the separation of concerns between id and position in the network. All peers maintain a partial routing table of the search space, that *negotiated* with the closest peers. Multiple peers can be responsible for the same path, resulting in a non uniqueness of routing and a robustness under peer failure.
- (Canon) is a multi-layer overlay where routing is based on a hierarchical DHT. As in Arigatoni, the search space is partitioned into *domains*; in contrast, routing inside a domain is DHT-based, and topology is static.
- (Coral) is another hierarchical DHT. The search space is partitioned into three *clusters*, based on latency; a regional cluster, a continental cluster and a planet-wide cluster. It also comes with algorithm for self-organizing, merging and splitting clusters, to ensure acceptable diameters.

Conclusions. In this paper, we describe the version V3 of the Arigatoni's generic resource discovery protocol. The new improved protocol RDP presented in this paper allows for *multiple instances*, *multiple services*, and *service conjunctions*. 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 V3 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 servers refuse those services .

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.

As part of our ongoing research, we are also working on a more complete mathematical study of our system, based on more elaborate statistical models and realistic assumptions, as well as the possibility to include 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. We are currently working on the implementation of a actual prototype and the subsequent deployment on the PlanetLab experimental platform (11), and/or on GRID5000, the experimental platform available at the INRIA (12).

Acknowledgment. We warmly thanks Pierre Lescanne and the anonymous referees for the useful comments and multiple constructive suggestions.

References

[1] D. Benza, M. Cosnard, L. Liquori, M. Vesin, Arigatoni: A Simple Programmable Overlay Network, in: Proc. of John Vincent Atanasoff International Symposium on Modern Computing, IEEE, 2006, pp. 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, LNCS, Springer, 2006, to appear. Also available as RR INRIA 5928.

[3] R. Chand, M. Cosnard, L. Liquori, Improving Resource Discovery in the Arigatoni Overlay Network, in: ARCS: International Conference on Architecture of Computing Systems, LNCS, Springer, 2007, to appear.

[4] M. Cosnard, L. Liquori, R. Chand, Virtual Organizations in Arigatoni, DCM: International Workshop on Developpment in Computational Models. To appear in ENTCS.

[5] E. Zegura, K. Calvert, S. Bhattacharjee, How to Model an Internetwork, in: Proc. of INFOCOM, IEEE, 1996, pp. 594–602.

[6] AEOLUS, Deliverable D2.1.1: Resource Discovery: State of the Art Survey and Algorithmic Solutions, Tech. rep., By Evangelia Pittoura, University of Ioannina, `http://aeolus.ceid.upatras.gr` (2006).

[7] H. Jagadish, B. Q. Vu, BATON: A Balanced Tree Structure for Peer-to-Peer Networks, in: Proc. of VLDB, ACM, 2005, pp. 661–672.

[8] K. Aberer, P-Grid: A Self-Organizing Access Structure for P2P Information Systems , in: Proc.of CoopIS, no. 2172 in LNCS, Springer, 2001, pp. 179–194.

[9] P. Ganesan, P. Krishna, H. Garcia-Molina, Canon in G-major: Designing DHTS with Hierarchical Structure, in: Proc. of ICDCS, IEEE, 2004, pp. 263–272.

[10] M. J. Freedman, D. Mazières, Sloppy Hashing and Self-Organizing Clusters, in: Proc. of IPTPS, no. 2735 in LNCS, Springer, 2003, pp. 45–55.

[11] Planet Lab Consortium, `http://www.planet-lab.org`.

[12] The Grid 5000 Consortium, `http://www.grid5000.org`.