CDC01-INV3104

A service network architecture for a multi-vehicle search mission

Marco Zennaro, Jeff Ko, Raja Sengupta and Stavros Tripakis

Abstract— Multi-vehicle applications rely on the dynamic allocation of resources such as vehicles, CPUs, bandwidth and storage, and must exhibit robustness to failures or to service degradation in general.
We present a model for such applications, called the *service network model* (SNM). The entities of this model are *services* and *service providers*. Services are defined by standard names and interfaces, and are described by attributes. Service providers *export* services with certain quality of service guarantees. They may also need to *import* services from other providers.
An application is modeled as a directed graph, where nodes represent service providers and edges represent services imported by the source node and exported by the destination node. The problem is then to build such application graphs dynamically, starting from logical descriptions, and reconfigure them appropriately in case of departures and arrivals of service providers.
We provide a middleware and an algorithm that solves the above problem. Functions of the middleware include publishing, finding and using service providers, as well as completing an incomplete application graph with the missing services.
We illustrate our approach with a case study involving a multi-vehicle search mission.
Index Terms—service network model, service network protocol suite, multi-vehicle search mission, distributed applications, control

I. Introduction

The growing ubiquity of communication networks is driving the development of systems of unprecedented size and heterogeneity. We have in mind systems such as unmanned air vehicle networks (see [1]), net markets (see [2]), metropolitan transportation management systems (see [3] and [4]), and so on.

These systems consist of many resources interacting to concurrently execute many tasks. More interestingly, the set of resources and tasks are dynamic. For example, in an unmanned air vehicle network, vehicles leave and join the network, routers go up and down, missions start and terminate. In a net market, buyers and sellers join and leave the network, and business transactions are constantly starting and completing.

In classical control system design the departure of a sensor or actuator would be treated as a failure. It is dealt with by designing another configuration to be adopted by the system in response to failure detection. Events such as the sudden joining of another actuator are inexpressible in traditional control design formalisms. In the systems of interest to us, such gross changes in the collection of sensors, actuators, or controllers are part of normal business. To treat each sensor departure as a failure to be accounted for by a new design is too complex. We shall also argue it is unnecessary.

We write this paper in pursuit of a new organizational principle for these heterogeneous, large-scale, multi-tasking systems. The principle described here is called a *service network* organization. Our objective is to find an organizational paradigm that will scale to systems with thousands of resources teaming for hundreds of tasks executing in concert over a geographically extensive theatre of operations. We also desire that the organization be robust or survivable in the sense of gracefully enhancing or degrading task execution in response to the arrival or departure of resources.

For inspiration we have turned to the organizational principles of the data transport Internet and the World Wide Web. Service networking seeks to organize large-scale systems in the way a routing protocol organizes routers. A network of routers is designed to execute packet delivery tasks. Each packet delivery task is specified by a source and destination address. The set of routers executing a packet delivery task emerges dynamically from an adaptive, distributed algorithm. Consequently, while routers fail, as long as there are sufficiently many routers and links, the packet delivery function survives. Likewise, as new routers appear, the allocation mechanism harnesses them to the packet delivery function without centralization or synchrony.

A service network is a network of service providers that offer services to each other and to clients of the network. Service providers might be compiled programs, servers, vehicles, databases, hosts, etc. A service network client requests services. The service network finds providers. Generally, the providers need other services themselves to function. Therefore they become service network clients in turn. The set of providers changes. The service network has resources and protocols to keep track of the set of current service providers and match them to the dynamically arising service requests.

The service network organization extends middleware like JINI [15] or CORBA [14] to achieve its objectives.

The paper is organized as follows. In Section II we present the Service Network Model formally and discuss the intuition behind it. We also state a problem of automatically building service network applications. In Section III we describe the middleware and algorithms we provide for solving the service network problem. In Section IV we illustrate our approach through a multi-vehicle search-mission case study. Section V concludes the paper.

II. SERVICE NETWORK MODEL

The Service Network Model (SNM) consists of two basic entities: *services* and *service providers*. A (service) provider *exports* a set of services, that is, it implements each service and makes it available to other providers. In general, for each service that it exports, a provider p needs to use a set of other services, exported by other providers. We say that p *imports* these services. A service network consists of a set of providers using the services of each other.

More formally, let *S* denote the set of services, and *P* the set of providers. A function *export:* $P \rightarrow 2^S$ maps each provider to the set of services that it exports. A function *import* maps a provider p and a service s in *export(p)* to a *multiset* of pairs of the form (S', P') where S' is a subset of S and P' is a subset of P, with the meaning that p needs *some* service in S' and this service should be exported by *some* provider in P'. This allows us to express constraints on the quality of service requirements that a provider importing a service imposes on the provider exporting it. The fact that *import(p,s)* is a multiset allows us also to express the fact that *p* may need two or more providers of the same service in order to operate, for example, a mission may require at least three vehicles.

A service network can be represented as a directed rooted graph *G=(V, r, E),* where *V* is a subset of *P, r* is the root node, and E is a subset of VxSxV. An edge *(p,s,q)* in *E* will represent the fact that p exports s, q imports s, and q uses the service s of p.

We say that G is *consistent* if $\forall$ (p,s,q) $\in$ G . s $\in$ export(p) $\wedge$ ($\exists$ (S',P') $\in$ import(q,s) . s $\in$ S' $\wedge$ p $\in$ P'). That is, if q uses service s of p, then indeed q must import s and p must export s.

We say that G is *complete* if $\forall$q $\in$ V . $\forall$ (S',P') $\in$ import(q) . $\exists$ s $\in$ S' $\wedge$ p $\in$ P . (p,s,q) $\in$ E. That is, all providers importing some service are indeed using some other provider that is exporting this service.

We say that *G* is *connected* if for every node *p* in *V* there is a path from *r* to *p.*

Given two graphs *G=(V, r, E)* and *G'=(V', r', E'),* we write $G \subseteq G'$ when $V \subseteq V' \wedge E \subseteq E'$.

The problem we are interested in can be stated as follows.

Service network problem (SNP) definition: Given an initial graph G_0, find a graph G such that:

1. $G_0 \subseteq G$.
2. G is consistent, complete and connected.
3. There exists no G' which is consistent, complete and connected, and such that $G_0 \subseteq G' \subset G$. In other words, G is a *minimal* solution.

In the following section, we provide an algorithm that solves the service network problem.

The above formulation abstracts away from details such as how exactly an interface is specified, what exactly is a service provider, and how does a provider uses a service exported by another provider. The answers to the above questions depend on the underlying implementation platform(s). Here, we give a few concrete examples of possible realizations. The following sections provide the details on our current choice of implementation, as well as a concrete example of using the above setting.

Services can represent anything from a Java *interface*, to a yellow-pages entry. Services may have *attributes* (included in *templates* in JINI) which can be assigned values upon requesting the service. For example, a *search* service might have attributes *precision* and *speed*, with values *high* or *low* and *fast* or *slow*, respectively. We can represent all these in our setting, by creating a separate service s in S, for each service with different attribute values (we could also incorporate directly attributes in our setting, and indeed need to do it in case the attribute values are infinite, but chose not to do it here for the sake of simplicity).

One important thing to note about services is that they have to be *standardized*, at least within the scope of a service network. Indeed, since there is no formal semantics associated with a service, only its name and attributes carry its meaning, in an conventional way. Therefore, *search* might mean different things in different contexts, and will probably have to be refined into something much more precise, for instance, *MissingPersonAerialSearch*. The same is true for the interfaces, where only the name of functions and arguments carries the meaning.

Similar needs have emerged in other contexts as well. For instance, in the RosettaNet project (see [5]) where every "Partner Interface Process" (i.e. every interaction between two entities) is standardized. In RosettaNet a XTM DTD fixes the interface and the syntax (see [6]) of every method, and the semantic is precisely described by an automaton and by a description.

We chose to abstract from the above issues of standardization, which we believe will be resolved by the industrial and technological needs. In our setting, it is assumed that when a provider exports s and another provider imports s, they “know” they are talking about the same service.

Service providers are *implementations* of services. They can represent physical resources such as a workstation, an intelligent vehicle, and so on, or logical resources, such as a compression filter or complex search algorithm, or combinations of both, such as a reliable transmission protocol running on a physical network.

Service providers may also have attributes, for instance, *cost*, taking values in USD per hour[1].

Again, for the sake of simplicity, we choose to represent attributes implicitly, by creating different “copies” of a provider, which can provide, say, the same service with different attributes at different costs. Such situations can be readily incorporated in our model without changing the results. The fact that the importer of a service may place restrictions on the exporters that it wants to use for that service, can be modeled in the function import. In the example above, if the importer wants to use only the inexpensive versions of the exporter of s, it will define the pair (S’,P’) such that P’ contains only the inexpensive “copies” of the exporters.

Although it may appear that the set of providers in the above setting is static, this need not be the case in practice. Indeed, it is possible to treat dynamic changes (e.g., failures) in the following way. Whenever some provider departs, this results in the graph G becoming incomplete. We can therefore invoke the algorithm again, starting from G’ (G without the failed node). If we manage to complete G’, then the failed provider has been replaced by some new one. If not, we might need to “roll back” and start from scratch (i.e., from G_0).

We are currently working on improvements on the above technique, where the “roll back” is limited to an autonomous subset of the graph.

III. Service Network Suite

In this section, we describe the Service Network Suite (SNS), which is a set of primitives we have built in order to facilitate the design and development of service network applications. These primitives allow service providers to “publish” themselves, as well as to search for and use other providers. The most elaborate primitive implements the algorithm to solve the service network problem formulated in the previous section. The suite uses the JavaSpaces [11,12] and Java technologies, and the algorithm to solve the SNP is currently written in Prolog. The structure of the suite is shown in figure 1.

We use the term *client* to refer to an entity (e.g., program) that is external to the service network, or to a service provider that uses the suite in order to find and use the services of other providers. The clients communicate with the suite via a Java API. Prolog engines process the requests. A number of JavaSpaces are used as knowledge bases that store the necessary information on the services in the network.

Service providers use the primitive *publish* to register with the suite, informing it of the services they implement and their attributes, if any. As a side tool, we have also developed a compiler, which automatically converts a Java class into a “publishable” service provider.

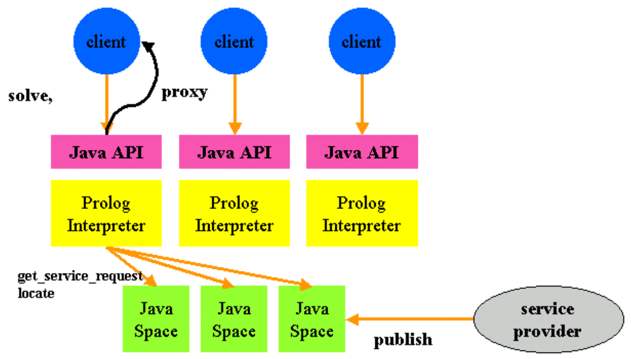

Fig. 1: service network suite architecture

There are five primitives offered to clients: *locate, notify, connect, create, solve* and *repair*.

Locate takes as input the specification of a service (with constraints on its attributes, if any) and returns all published providers that export this service with these attributes.

Notify takes the same input as *locate*, and results in notifications being sent to the caller whenever a service provider offering the specified service is published.

Connect allows the importer of a service to be connected to the exporter of this service. After being connected, the importer can start using the service.

For load-balancing purposes, our implementation distinguishes between two types of service providers: *running* and *dormant*. Running providers are implemented as servers that run constantly, waiting for clients to use their service. Dormant providers are mere implementations of a service (e.g., Java classes implementing an interface), but they need an execution platform in order to run.[2] A client can recognize whether a provider is dormant or not, and in case it is, the client uses *create* to spawn a temporary active version of the provider. The suite is responsible for finding free computation resources where the newly spawned provider is to be run, and also for freeing the resources when the provider is not used anymore.

Solve takes as input an initial graph G_0 and solves the SNP described in the previous section.

[1] It is important to standardize the particular meaning and semantic of the attributes in order to avoid what is happened with the last Polar Mars Lander (see [9]).)

[2] In our case, this execution platform is in fact implemented as a provider itself, offering the *run* service.

Solving the SNP means that a graph G is found, the providers in this graph are *connected*, and dormant providers (if any) are *created*.

Repair relies on *solve* to complete a graph whenever a provider departs, as discussed in Section II.

We provide a language, the Service Network Language (SNL) to specify requests (i.e. a graph like G_0). A request in SNL is defined by the BNF grammar in figure 2.

```
user_request ← '(' command_list ,
                   topological_constraint_list ')'

command_list ← command, command_list
command_list ← nil

command ← 'import' '(' importer_provider,
          service, exporter_provider, service_constraint_list,
          exporter_constraint_list ')'

importer_provider ← provider
exporter_provider ← provider

provider ← provider unique identifier
provider ← __
service ← service unique indentifier
service ← __

service_constraint_list ← service_constraint,
                          service_constraint_list
service_constraint_list ← nil

provider_constraint_list ← provider_constraint,
                           provider_constraint_list
provider_constraint_list ← nil

service_constraint ← comparator  '(' service,
                     attribute, value ')'
provider_constraint ← comparator '(' provider,
                      attribute, value ')'

attribute ← string
comparator ← 'equal' | 'lesser' | 'greater'
value ← integer | string | ……

topological_constraint_list ←
                      topological_constraint,
                      topological_constraint_list
topological_constraint_list ← nil

topological_constraint ← 'uses' '('
                      importer_provider, exported_service,
                      exporter_provider, service_constraints_list,
                      provider_constraints_list ')'
```

Fig. 2: a BNF for the user and services request in SNL

In the service network problem a request is a graph G_0 as stated in the previous section. $G_0 = (V, r, E)$ can be mapped into a SNL request (commands, topological_constraints):

(P,S) ∈ import(r,) ⇔*
import(Importer, Serviceid, Exporter, Service_constraints, Exporter_constraints) ∈ commands ∧
P = subset($\mathfrak{S}$, serviceid, service_constraints) ∧
S = subset($\mathfrak{P}$, exporter, exporter_contraints)

(P,S) ∈ import(t,), t ∈ V, t ≠ r ⇔*
uses(Importer, Serviceid, Exporter, Service_constraints, Exporter_constraints) ∈ topological_constraints ∧
S = subset($\mathfrak{S}$,serviceid, service_constraints) ∧
P = subset($\mathfrak{P}$,exporter, exporter_constraints)

∀ providers_set ∈ V . ¬∃ (importer, service, exporter) ∈ E . exporter ∈ providers_set ⇒
uses(__, __, __, __, provider_constraints) ∧
provider_set = subset($\mathfrak{P}$,exporter, exporter_constraints)

where $\mathfrak{S}$ is the set that contains all the services, $\mathfrak{P}$ is the set of all the providers and the *subset* operator extracts a subset of $\mathfrak{S}$ (resp. $\mathfrak{P}$), given the service (resp. provider) unique identifier and a set of constraints on its attributes. In the SNL language subsets are not enumerated but described using identifiers and attributes.

Example: In the multi-vehicle search scenario (see Section IV), the initial graph consists of a single provider MC (mission control), exporting the service *mission,* and importing a *search* service, without any attribute constraints:

G_0 = {(MC)}

Import(MC, *mission*) = ($\mathfrak{Search}$, $\mathfrak{P}$)

where $\mathfrak{Search}$ is the set of *search* services. Figure 3 illustrates this graphically.

In SNL, this request will be described as shown in figure 4.

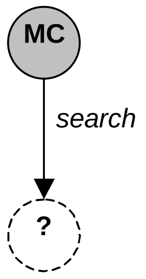

Fig. 3: the request in the SN model

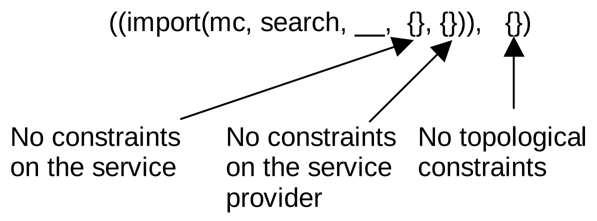
((import(mc, search, __, {}, {})), {})
No constraints on the service
No constraints on the service provider
No topological constraints

Fig. 4: the request expressed in SNL

Apart from the "user request", that is, the initial graph plus topological constraints, each provider can have its own requirements, for each service that it exports. These requirements are called *service* *requests*, are expressed in SNL, and are deposited by the provider when it publishes the service.

The algorithm that implements *solve* is written in PROLOG. A simplified version of this algorithm is given in figure 5. This is not the most efficient way to implement *solve,* however, we omit the more elaborate version for the sake of readability.

```
Solve(Request, Solution) |-
            complete(Request, [], Solution ).

Complete( (Import_list, Constraint_list),
     Partial_Solution, Solution) |-
             process_import_list(Import_list, {}, Partial_solution,
                                         Solution),
             process_constraints(Constraint_list, Solution).

process_import_list([], __, X,X,).
process_import_list([import(Importer, Service, Exporter,
     Service_constraints, Exporter_constraints) | []], Exporters_list,
     Partial_Solution, Solution) |-
            process_import(Importer,Service, Exporter,
            Service_constraints, Provider_constraints,
            Partial_solution, Solution),
            not member(Exporter, Exporter_list).

process_import_list([import(Importer, Service, Exporter,
     Service_constraints, Exporter_constraints) | T], L,
     Partial_Solution, Solution) |-
            T isnot [],
            process_import(Importer,Service, Exporter,
            Service_constraints, Provider_constraints,
            Partial_solution, Improved_partial_solution),
            not member(Exporter_Exporter_list),
            append((Exporter,Service), Exporter_list, New_list)
            process_import_list(T, New_list,
            Improved_partial_solution,  Solution)

process_import(Importer,Service,Exporter,Service_constraints,
     Exporter_constraint, Partial_solution, Solution) |-
             member((__,Service,Exporter)),
             check_service(Service, Service_constraints),
             check_provider(Exporter,Exporter_constraints),
             append(Importer,Service, Exporter), Partial_solution,
             Solution).
process_import(Importer,Service,Exporter,Service_constraints,
     Exporter_constraint, Partial_solution, Solution) |-
             locate(Service,Provider_list,Service_constraints,
                     Exporter_constraints),
             member(Exporter, Provider_list),
             append((Importer,Service,Exporter),Partial_solution,
                      Improved_partial_solution),
             unfold(Service,Exporter,Improved_partial_solution,
                      Solution).

unfold(Service,Exporter,Partial_solution,Solution) |-
             get_service_request(Service,Exporter,Request),
             complete(Request,Partial_solution,Solution)

process_constraints([], __).
process_constraints([uses(Importer,Service, Exporter,
     Service_constraints, Provider_constraints) | T], Solution) |-
         member((Importer,Service,Exporter), Solution),
         check_service(Service, Service_constraints),
         check_provider(Provider,Provider_constraints),
         process_constraints(T)
```

Fig.5: a simplified algorithm for the service network problem

Solve gets a request and unifies Solution with a consistent, complete, minimal and connected graph that satisfies the user request. The PROLOG program uses the JavaSpaces, in order to locate service providers and get their service requests. The two corresponding PROLOG clauses are *locate* and *get_service_request*. The first unifies the Provider_list with the list of providers that offer the specified services and satisfy the constraints.

get_service_request is used to retrieve from the knowledge base the service request deposited when a service is published.

The program given in Figure 5 also handles multiple requests of the same service, making sure the providers satisfying these requests are different.

Example: We end this section by a simple example that illustrates the execution of our algorithm.
Suppose that the knowledge base contains the entities in figure 6 (the attributes are not in the table in order to simplify the problem):

Provider	Service	Service Request
p1	search	{{import(p1,sweep,__,{},{}}, {}}
h1	sweep	{{},{}}
h2	sweep	({},{})

Figure 6: a simple knowledge base

Let us suppose that the client calls *solve* on the request of figure 4:

solve(((import(mc, search, __, {}, {})),   {}), Solution)

The PROLOG interpreter will try to satisfy this predicate using the only *solve* rule. It will unify the variable Request with the user request and will try to satisfy:

complete(Request,[], Solution))

At this point the *complete* rule is used. The empty set is unified with the partial solution and it will try to satisfy the two sub-goals:

process_imports(Import_list, {}, Partial_solution, Solution),
process_constraints(Constraint_list, Solution).

Since the request does not contain any topological constraints, the second clause is satisfied because of the fact process_constraints([], __). As for the first clause, since Exporter_list is empty the not member subgoal is trivially true. The interpreter will use the second process_import_list rule to satisfy the subgoal since the first one does not apply.

process_import(mc,search,Exporter,{}, {}, {}, Solution)

At this point the interpreter will try the first process_import rule: it will check if there are some providers in the partial solution built so far (i.e. the empty set) that offer the requested service. This rule is necessary to ensure minimality. Since this rule fails, the second rule is tried:

locate(search,Provider_list,{},{}),
member(Exporter, Provider_list),
append((mc,search,Exporter),{},
Improved_partial_solution),
unfold(search,Exporter,{}, Improved_partial_solution).

The locate expression is always true and links the PROLOG interpreter with the knowledge base. The provider list is unified with the list of providers that offer the search service in the network (in this example {p1}). The second expression unifies Exporter with p1. Then a new partial solution is built. We add the edge (mc, search, p1) to our solution. The unfold rule is called to ensure completeness. The only unfold rule is then used to satisfy it:

get_service_request(Service,Exporter,Request),
complete(Request,Partial_solution,Solution)

The first expression links PROLOG with the knowledge base. It returns the service requests that were deposited at the moment of the publication. The clause *complete* is called to complete the sub-graph rooted in the provider of the *search* service. We have reduced the problem by one level.

Similarly *complete* will be satisfied using the *process_import_list* second rule, that will in turn be satisfied using the *process_import* third rule, since there are no services in the network that offer the *sweep* service.

locate(sweep,Provider_list2,{},{}),
member(Exporter2, Provider_list2),
append((p1,sweep,Exporter2),{},
Improved_partial_solution),
unfold(sweep,Exporter2,{},Improved_partial_solution).

Now, *locate* will unify Provider_list2 with {h1,h2}. *Member* will unify Exporter2 with h1. If the interpreter rolls back (i.e. if there is something wrong with h1) it will unify Exporter2 with h2. The partial solution is improved. A new edge is added: (p1,sweep, h1). The *unfold* sub-goal ensures completeness. It is satisfied by expanding these two sub-goals:

get_service_request(sweep,h1,Request),
complete(Request,Partial_solution,Solution)

The *sweep* service does not rely on any service and so its request is a couple of empty lists. The complete is satisfied by these two sub-goals:

process_imports({}, Partial_solution, Solution),
process_constraints({},, Solution)

The second is trivially true. The first is true and is the tail of the recursion. Partial_solution is unified with Solution by the factprocess_import_list({}, X,X).

The final solution is shown in figure 7.

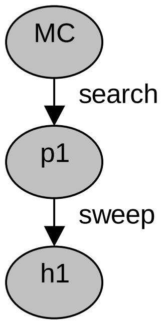

Fig.7: the solution of the first example generated by our algorithm

IV. CASE STUDY: A MULTI-VEHICLE SEARCH

The scenario is as follows: an enemy robot has invaded the territory and mission control (MC) wants to locate it. For that, MC relies on a *search* service. MC provides the coordinates of the area to search and a description of the enemy robot.

The *search* service divides the territory into sectors. It coordinates multiple vehicles based upon what they currently see and what they found in the past. We currently use a random search algorithm (a more intelligent algorithm is given in [10]). The *search* providers rely on *sweep* services and on a *reliable data storage* to share information (a probability map for the intelligent search).

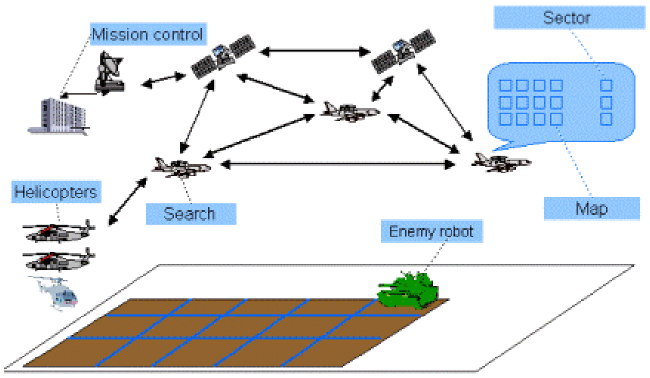

Fig. 8: the multi-vehicle search scenario

The helicopters offer the *sweep* service. They receive the coordinates of the area to sweep and report their findings to the reliable storage provider.

It is straightforward to adapt this mission to other types of vehicles: the only requirement is that the new vehicles also export a *sweep* service. The remaining components of the network do not need any modification.

The reliable storage stores all information collected by the sweepers and computed by the search service. If the search service crashes the mission does not need to be restarted since all information is reliably stored. Currently, we use a single JavaSpace as a reliable storage provider.

The *Transaction Manager* service is used by the JavaSpace and by the sweepers to maintain data consistency.

The service network looks has the structure shown in Figure 9. The nodes with a "?" correspond to the unknown service providers that the *solve* primitive is supposed to fill in. This is done by giving the user request and the service requests of each provider.

We first give the service requests in SNL.

The provider of the *search* service, call it *sp,* publishes the following:

Service Name: *search*
Service Request*: ({import(sp, sweep, __,{},{}),*
import(sp, sweep, __,{},{}),
import(sp, sweep, __,{},{}),
import(sp, reliable_storage, __,{},{}},
{})

Each provider of *sweep* service publishes:

Service Name: *sweep*
Service Request:*({import(self,reliable_storage, __),*
import(self,transaction_manager,__)},
{})
where *self* denotes the specific provider.

A reliable storage provider publishes:

Service Name: *reliable_storage*
Service request: *({*
import(self,transaction_manager,__,{},{})},
{})

A transaction manager service provider publishes:

Service Name: *transaction manager*
Service Request: *({},{})*

The mission-control request that triggers the construction of the service network will be:

({import(mission_control,search,__,__,{},{}},{})

With this request, MC is not interested in how the *search* service is provided. If it wants to specify an exact set of helicopters, for instance {h1, h2, h3}, that must be used to conduct the search, then the request will be:

({import(mission_control,search,__,__,{},{}},
{uses(__,__,h1,{},{}), uses(__,__,h2,{},{}),
uses(__,__,h3,{},{})})

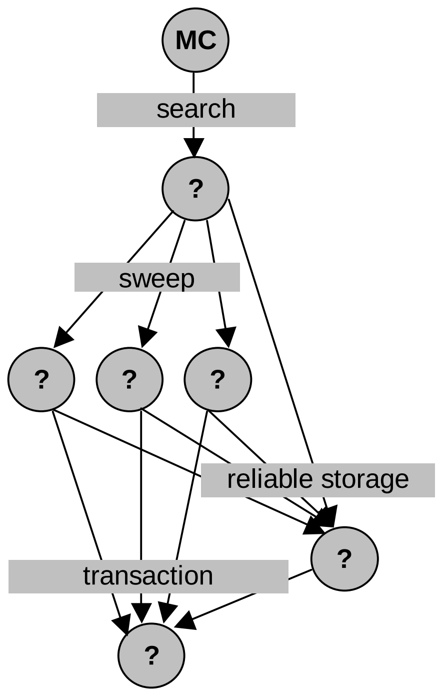

Fig. 9: the NS model we developed for the multi-vehicle search

V. Conclusions and Future Work

We have presented a model called Service Networks which we believe is appropriate for dynamic heterogeneous distributed applications, in particular, those involving multi-vehicle operation scenarios. We have developed a middleware based on Java and JavaSpaces technology, and an algorithm, for automatically building and reconfiguring service networks. We have illustrated our approach with a multi-vehicle search-mission scenario.

A lot of work remains to be done, in particular in terms of algorithms. Our Prolog implementation given here is simple and understandable enough, but not very efficient. The algorithms can be specialized and implemented in far more efficient ways in an imperative language. Also, the *repair* method is currently far from satisfactory. We are currently working on improving it, by limiting the "roll back" to autonomous subsets of the incomplete graph.

Another issue involves the execution of multiple *solve* processes in parallel. This is inevitable if we want our approach to be scalable. The issue that arises in that case has to do with the concurrent allocation of resources. Indeed, some providers may represent physical resources that cannot be replicated. Therefore, some *concurrent reservation* mechanism is necessary, so that such a provider can participate in the solution of multiple instances at the same time, although it will be allocated to at most one solution at the end. As a straightforward rule, the provider can be *locked*, so that it can be used by at most one instance at a time. However, more efficient techniques are needed in general.

REFERENCES

[1] http://robotics.eecs.berkeley.edu/bear/
[2] "The B2B Internet Report: Collaborative Commerce", C. Phillips, M. Meeker, Morgan Stanley Dean and Whittier, April 2000
[3] http://transacct.eecs.berkeley.edu
[4] "Freeway performance measurement system (PeMS) shows big picture", Chen, Chao; Jia, Zhanfeng; Petty, Karl; Shu, Jun; Skarbardonis, Alexander; Varaiya, P. P., Intellimotion. Vol. 9, no. 2, pag. 8-8,12, 2000
[5] www.rosettanet.org
[6] http://www.w3.org/XML
[7] SHIFT Tutorial: A first course for SHIFT programmers, Tunc Simsek, http://www.eecs.berkeley.edu/~simsek
[8] A theory of hierarchical, distributed systems, P. Varaiya and T. Simsek, ONR review, Los Angeles, CA. July 1998.
[9] http://mars.jpl.nasa.gov/msp98/lander
[10] Hespanha, Kim, Sastry "Multiple-Agent Probabilistic Pursuit-Evasion Games", Proc. Of the 38th Conf. On Decision and Contr., Dec 1999
[11] Freeman, Hupfer, Arnold "Javaspaces Principles, Patterns, and Practice" Sun publications, Addison-Wesley 1999
[12] http://www.sun.com/jini/specs/jini1.1html/jsTOC.html
[13] Wolfson, Jajodia, Huang "An adaptive Data replication Algorithm", ACM Transactions on Database Systems, Vol. 22, No. 2, June 1997, Pages 255-314
[14] http://www.corba.org
[15] http://www.jini.org