Group Orchestration in a Mobile Environment

Eline Philips, Jorge Vallejos, Ragnhild van Der Straeten, Viviane Jonckers

Group Orchestration in a Mobile Environment

Eline Philips**, Jorge Vallejos, Ragnhild Van Der Straeten, and Viviane Jonckers

Software Languages Lab, Vrije Universiteit Brussel, Belgium
ephilips,jvallejo,rvdstrae@vub.ac.be vejoncke@soft.vub.ac.be

Abstract. The increasing popularity of mobile devices fosters the omnipresence of services in mobile environments. Software systems in a mobile environment often want to manage a set of services that form a logical group and orchestrate the execution of a particular process for all its members. To orchestrate a group of services, abstractions are required which allow control over the execution in a way that transcends the individual process of a single member. Currently, existing languages do not offer adequate abstractions to perform said group orchestration in a reliable way. In this paper we present high-level abstractions for group orchestration as a new set of workflow patterns. We show how these patterns are integrated in an existing workflow language for nomadic networks, i.e. NOW. The workflow language NOW handles network and service failures at the core of the language. By extending this fault tolerance to the new group abstractions, we show how to conduct these in a reliable way.

1 Introduction

People everywhere are surrounded by a wide range of mobile and stationary devices that can perform all kinds of services. For instance, today's smartphones are able to determine the temperature, location, orientation, etc. Since devices and services are everywhere nowadays, there is a need to address several services or devices at the same time. For instance, to determine the current temperature at a certain location, one could retrieve that information from several temperature services in the neighbourhood and calculate the average.

Existing approaches that interact with a group of services can be either classified in the domain of group communication or group behaviour. *Group communication* [14] addresses technologies that can enable effective communication between various groups in the network, using for instance multicasting. *Group behaviour* [12] is the capability of services to coordinate with each other. In order to coordinate a group of services, the dependencies between the members of the group must be managed in order to let them collaborate.

These domains focus on communication and/or coordination, but have little support for orchestration. *Service orchestration* is defined by Peltz [15] as a

** Funded by a doctoral scholarship of the "Institute for the Promotion of Innovation through Science and Technology in Flanders" (IWT Vlaanderen).

business process that interacts with both internal and external Web services. We define *group orchestration* in a mobile environment as the management of a set of services that form a logical group where all its members execute a particular process. Moreover, there is a need to control the execution of the group members in a way that transcends the individual process. Group orchestration should be able to deal with both the voluntary and involuntary removal and addition of group members. It is also essential that there are ways to synchronise and streamline the execution process of several group members.

As group orchestration provides high-level abstractions to manage the execution of processes for the group members, it differs from group communication which is only concerned about the low-level protocols that can be used for the underlying communication. Group orchestration also differs from group behaviour. Even though group behaviour also needs to take the necessary precautions to handle unforeseen network failures, it focusses on the collaboration between the group members unlike group orchestration which aims at the management of the execution of a process by all the members of the group.

In this paper we present high-level abstractions for group orchestration and introduce these abstractions in the workflow language NOW. NOW is a workflow language that allows the orchestration of distributed services in a mobile network. This workflow language introduces high-level control flow patterns and abstractions which can deal with network failures that are inherent to a mobile environment. We extend this workflow language with novel abstractions to enable reliable group orchestration in a mobile environment.

The paper is organised as follows: in Section 2 we give a motivating example and an enumeration of the requirements the proposed abstractions need to fulfill. In Section 3 we give a short introduction to NOW as we extend this language with novel patterns for group orchestration. In Section 4 the novel abstractions for group orchestration with services are presented. Before concluding the paper, we describe related work.

2 Motivation

In this section we describe an example scenario which emphasises requirements for group orchestration in a mobile environment.

"The headliner of the Pukkelpop festival decides to surprise its fans with a special concert by letting them vote for the songs that will be played. In order to accomplish this they use the festival's infrastructure to communicate with the mobile phones of the fans who are present in the festival area. All fans who are interested in participating in this vote receive a list of the band's discography. They are able to vote until two hours before the band's concert is scheduled. All votes that are received afterwards are considered invalid. As a special bonus, the voters have the benefit of receiving the band's final playlist before the start of the concert."

We enumerate the requirements for group orchestration that can be distilled from this small example. These requirements for group orchestration in a mobile environment can be divided into three categories, namely "definition of group membership", "synchronisation mechanisms", and "failure handling".

1. [**C1**] **Definition of Group Membership**:
   - ***Intensional definition*** [***R1***]: Users should be able to define processes that will be executed by a set of services that form a logical group. This group can be either defined extensionally, by enumerating all its members, or intensionally by giving a description all members must fulfill.
     In the example scenario, a group is defined intensionally, namely all fans of the headliner who are located in the festival area.
   - ***Plurality encapsulation*** [***R2***]: First of all, users want to orchestrate a group of services as if they are one single unit. Moreover, as we are targeting mobile environments, this quantity of group members can fluctuate over time as new services join and disjoin the group. This requirement is known in existing literature as the need for encapsulating plurality [11] or arity decoupling [10].
     As we show in the scenario, the members of the group are not known a priori and can change over time. For instance, fans can arrive at the festival area at a later point in time then other fans who were there earlier and who have already received the request to vote.
   - ***Dynamic modification*** [***R3***]: During the execution of the group it must be possible to redefine the members of this group. It should be possible to restrict the members by filtering out members based on a certain condition and also to change the group's description causing the arity of the group to change.
     In the motivating example, the group initially consists of all fans of the headliner. Later on, only those fans who are interested in voting are being addressed.
2. [**C2**] **Synchronisation Mechanisms** [**R4**]: In order to streamline all execution processes of the group members, the fact which service can do which task at what time needs to be managed and controlled. Moreover, the number of times a specific task is executed and the data needed during this execution should be controllable. Synchronisation mechanisms can let processes wait, redirect and even abort in order to let given criteria persist. This way, synchronisation mechanisms influence the amount of members of the group.
   In the example scenario, all results need to be gathered two hours before the headliner's concert is going to start. This task should only be performed by a single service at a specific point in time. Therefore, all data needs to be collected before the execution of that task can start.
3. [**C3**] **Failure Handling** [**R5**]: As mobile environments are liable to volatile connections, ways to detect and handle failures must be available. First of all, it should be possible to react upon a failure that occurs during the individual process execution of a single member of the group. Moreover,

there must be mechanisms to detect and handle failures at the group level and even propagate individual failures to the group level.
In case a failure occurs within an individual process execution of a single fan, there should be a compensating action that tries to re-execute the process (for instance, resending the message). However, when something goes wrong when the votes of all fans are being gathered, the compensation should apply for all fans, hence the entire group's execution.

3 Small Introduction to NOW

In this section we give a brief introduction of the workflow language NOW, as this is the language we extended with novel abstractions to support group orchestration. NOW [5] is a workflow language sculpted for *nomadic networks* [1]. These networks consist of a fixed infrastructure and mobile devices that try to maintain a connection with that infrastructure. These kinds of networks are omnipresent, for instance, airports, shopping malls, hospitals, etc.

NOW is built as a library on top of AmbientTalk, a distributed scripting language targeting mobile ad hoc networks [3]. For both types of mobile networks, disconnections are considered the rule rather than the exception. Hence, in order to ensure that the workflow description cannot become unavailable during its execution, the workflow description resides on the fixed infrastructure of the network.

Failure Handling NOW introduces control flow patterns and failure handling patterns. The control flow patterns that are supported by the language range from very basic ones, like sequence and parallel split, to more advanced ones, like multiple instances patterns. NOW's *failure* pattern wraps a subworkflow and specifies compensating actions for possible failures that can occur. The different kinds of failures that can be detected are: a service that cannot be found, a disconnection that occurred, a timeout that occurred while interacting with a specific service, or an exception raised by that service. For these failures, one can specify compensating actions like restart, retry, rediscover, skip, wait, or executing an alternative subworkflow. These compensating actions can be chained together, to support compensations like *wait 20 seconds, retry to invoke the same service twice, when this fails execute another subworkflow.* As volatile connections are inherent to the type of networks targeted by NOW, the language supports default compensating actions.

Data Flow NOW also uses a dynamic data flow mechanism that passes data between the distributed services in the environment by employing a *data environment*. Such a data environment is a dictionary, containing variable bindings, that is passed between the activities of the workflow. Each activity can specify input and output variables used for its service invocation. The output variables (specified by the `@Output` annotation) cause (new) variable bindings to be added to

the data environment. In case of fork patterns (like parallel split), this data environment is conceptually copied and when a join occurs (like a synchronization pattern), these incoming data environments are merged using a merging strategy (like choosing the one with a maximum value for a certain variable). For more detailed information we refer the interested reader to [5] and [6].

4 Abstractions for Group Orchestration

In this section we describe the novel abstractions we introduce in NOW in order to satisfy the requirements for group orchestration we presented in Section 2. The description of these abstractions adheres to the categories and requirements we presented in Section 2. The implementation of NOW is extended with these novel abstractions and is available at [13]. In the remainder of this section we use small excerpts from the motivating example to illustrate their use. The entire implementation of this example scenario can also be found at [13].

4.1 Definition of Group Membership

First of all, we need to introduce an abstraction for a group, namely a subworkflow that must be executed for a set of services. In order to do so, we extended NOW with the notion of a *group* pattern.

```
Group( <description>, <variable name>, <subworkflow> )
```

This pattern is instantiated with a description of the services, a variable name that can be used to refer to the member services individually, and the subworkflow that must be executed several times.

Intensional Definition [R1] The description of the services can be either achieved by enumerating all of them (extensional description) or by describing all properties those services must fulfill (intensional description). Deciding which services to interact with in a dynamically changing environment is hard when reasoning extensionally about it, as the set of services can vary over time. In these kinds of environments it is opportune to provide intensional descriptions for those services. Intensional descriptions abstract away the precise number of services during interaction and let services maintain anonymity during this interaction. A group should be able to specify a description such that not only services of the same type, like temperature services, but also more sophisticated characterisation of members can be achieved. In particular, there is a need for intensional descriptions of services such as "*the service I last used*", "*my favourite service*", or "*all temperature services that are nearby and have an accuracy of more than 95%*". In order to support such intensional descriptions, the logical coordination language CRIME [4] can be used.

The Fact Space Model [4] of CRIME provides a logic coordination language for reasoning about context information that is represented as facts in a federated

fact space. Concretely, facts are locally published by applications and transparently shared between nearby devices as long as they are within communication range. Applications have the ability to react upon the appearance of facts, by making use of rules. This logic language uses the forward chaining strategy for deriving new conclusions as this data-driven technique is very suitable for the event-driven nature of CRIME. Therefore, we integrated CRIME in NOW which is targeted towards nomadic networks where lots of events (connections, disconnections, etc.) occur.

We support both types of group descriptions: extensional and intensional. First of all, it is possible to instantiate a group with an array of objects. These objects can be either references to the services (e.g. the fans of Pukkelpop's headliner), or just plain objects (for instance all ids of those fans). Additionally, we allow intensional description of the group members by either using a *type tag*[1] or by writing a logical expression in CRIME.

In our example scenario, all the fans of Pukkelpop's headliner need to be addressed. This can be expressed by writing a logical expression in CRIME[2]:

```
pp_visitor(?id), band_info(?band, "headliner"), fan_info(?id, ?band)
```

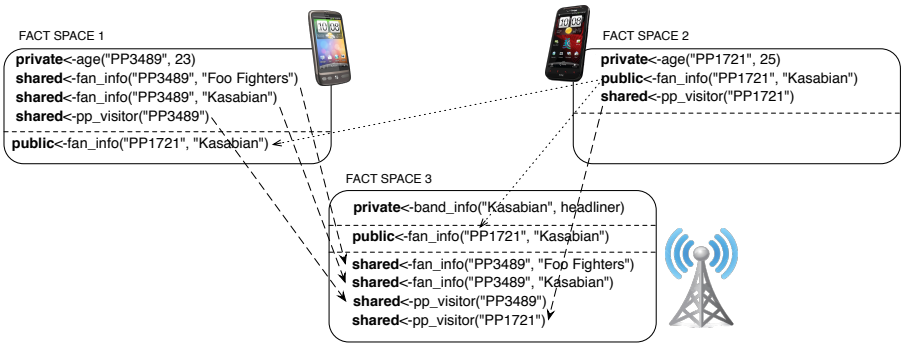


Fig. 1. Federated fact spaces of co-located devices.

In Figure 1 we depict the federated fact spaces of two fans and the federated fact space residing on the festival's infrastructure who are connected in a mobile ad hoc network. Those fact spaces consist of *qualified facts* which denote the fact space the fact belongs to. The different kinds of fact spaces we support are "private", "public" and "shared".[3] Private facts are not exchanged between co-located devices, whereas public ones are. In order to limit the exchange of facts between co-located devices, we added a third type of fact space "shared".

[1] AMBIENTTALK's type tags, wrapped JAVA interfaces, represent service types.
[2] CRIME's syntax resembles the one of PROLOG, although CRIME is a forward chainer.
[3] The shared fact space was added when integrating CRIME in NOW.

Facts that are published in this fact space are only exchanged to fact spaces who have subscribed to that type of facts. In our example, the infrastructure of the festival is interested in a lot of information of the fans, for instance the facts of type `pp_visitor` and `fan_info`, whereas fans are, in general, not interested in those published facts of other fans. Therefore, the facts that are published in the shared fact space are only asserted in the federated fact space of the infrastructure (`fact space 3` in Figure 1).

Plurality Encapsulation [R2] A group pattern can be started by executing its `start` method with its incoming data environment, the same way it is done for other workflow patterns in NOW [6]. When the group pattern is started, first all services satisfying the group's description need to be retrieved. In case of an extensional description, there is no need to query the backbone, but for intensional descriptions the services satisfying the description need to be looked up. Once these services are retrieved, the incoming data environment is cloned for each of these services. This way, each member of the group has his own data environment where local changes can occur. In order to access the specific service (member) for which the subworkflow is executed, a reference to the service is added in the data environment used to start each individual instance. When an intensional description was used to define the group members, the variable name (<variable name>) is bound to an AmbientTalk *far reference*[4] to the particular member. Afterwards, the subworkflow that is wrapped by the group pattern, is started with each of these data environments, as can be seen in Figure 2.[5]

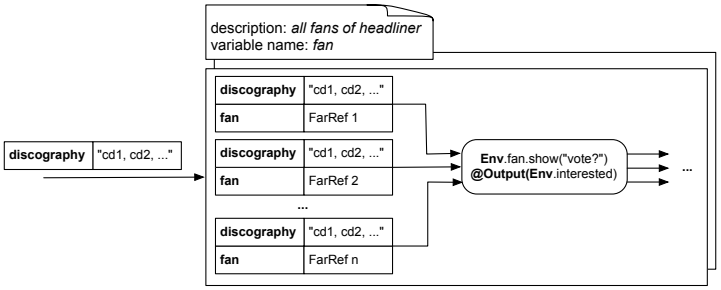


Fig. 2. Starting the execution of a group pattern.

Once the group pattern is started, it is possible that the backbone discovers a new service that satisfies the group's description. In this case, a new member is added to the group and the subworkflow wrapped by the group is started once

[4] A far reference is AmbientTalk's remote object reference.

[5] This diagram and the other diagrams in the remainder of the paper are used to help the reader follow the examples we present.

more. Note that this only applies when an intensional description (type tag or CRIME expression) was given. However, sometimes this behaviour is not wanted. Therefore, NOW also allows the specification of a *snapshot group*, where the number of services communicated with is fixed.

```
SnapshotGroup( <description>, <variable name>, <subworkflow> )
```

Unlike a normal group, such a snapshot group does not allow new members to join the execution once started. Note however that once the snapshot group is made, members of the group can still disconnect. By using NOW's failure handling mechanisms, it is possible to react upon such a disconnection of a group member in an appropriate way. This is described in more detail in Section 4.3.

Dynamic Modification [R3] As we already mentioned, it should be possible to redefine the members of a group when its execution is going on. For instance, it should be possible to restrict the members of the group by filtering out those members that do not satisfy a certain condition. Therefore, we introduce a *filter* pattern which only allows the instances who satisfy the given condition[6] to continue their execution.

```
Filter( <condition> )
```

In the motivating example of Section 2, the group is initially executed for all fans of Pukkelpop's headliner. However, after being asked if they are interested in participating, the members of the group are restricted to only those whom expressed their enthusiasm. This is depicted in Figure 3, where the condition of the filter will verify the value of the variable `interested` in the data environment, `Env`.

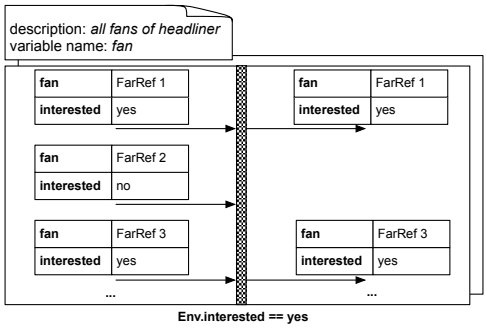


Fig. 3. Restricting the members of the group by using a filter.

[6] In Section 4.2 we give more details about the kinds of conditions that can be used to instantiate a filter with.

4.2 Synchronisation Mechanisms [R4]

The introduction of group orchestration gives rise to more advanced synchronisation patterns. The inherent volatile connections of the network cause communication partners to disconnect making full synchronisation not always possible. For instance, synchronisation should be able to succeed when only partial results are returned (after the first result, after a number of results, after some time, etc.). In this section we elaborate on the different synchronisation mechanisms needed to orchestrate a group in NOW. All of these synchronisation mechanisms do not only have an influence on the execution of the subworkflow by each of the group members, they are also reflected on the group itself. We present several types of synchronisation mechanisms that can be used to streamline the execution of the group's instances.

In this section we describe four patterns that enable group synchronisation, namely a *barrier*, *cancelling barrier*, *group join*, and *synchronised task*. Those patterns have some criteria in common:

- A condition is given to specify *when* the synchronisation may succeed. All individual instances whose execution was blocked, can continue their execution the moment the condition is satisfied.
- When the given condition is fulfilled, it is possible that next instances (i.e. instances that reach the synchronisation at a later point in time) should not continue their execution. Therefore, it should be possible to state whether or not a *cancellation* should take place.
- It is possible that when the condition is fulfilled, a specific task (subworkflow) needs to be executed *once*.
- When such a one-time task is specified, a merging strategy can be defined to specify which *data* must be available during the execution of that task.

In the remainder of this section, we first elaborate on the conditions that are used to instantiate group synchronisation patterns. Thereafter we describe four specific synchronisation patterns which use some of the criteria mentioned above.

Conditions used by Group Synchronisation Patterns All group synchronisation patterns are instantiated with a certain condition, such as *"after 10 seconds"*, *"when all instances have succeeded"*, or *"when 90% of the instances have succeeded"*. Such a condition can be either classified as a *time constraint*, a *quota constraint*, or a combination of both. NOW supports two different kinds of time constraints, namely a deadline (`At`) and a duration constraint (`After`). The *at* condition is fulfilled at a certain moment in time, whereas *after* succeeds a predefined time after the synchronisation pattern is reached for the first time. NOW also supports two kinds of quota constraints: `Percentage` and `Amount`. The *amount* and *percentage* condition take as argument a number and are satisfied when that number, or that given percentage of instances respectively, has reached the pattern. The above conditions can be combined using logical expressions and a *combiner* pattern. For instance the expression `Combiner( or(`

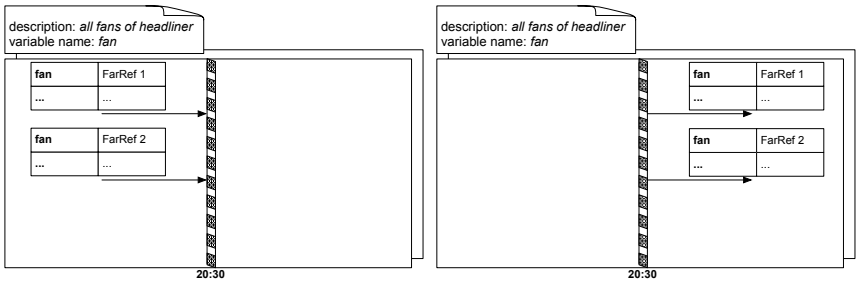
`Amount(1), After(60) )` implements a condition that is satisfied *"when one instance has started the synchronisation pattern, or after 60 seconds"*. It is also possible to extend the quota constraints with a user-defined function.

Barrier and CancellingBarrier Pattern We now present two novel patterns that allow synchronisation of individual instances of group members by blocking their execution until a specified condition is fulfilled.

```
Barrier( <condition> )
CancellingBarrier( <condition> )
```

When the execution of an individual instance of a group member reaches a *barrier*, the condition of the pattern is verified. When the condition is not yet fulfilled, the execution of that instance is blocked. At the moment the condition is fulfilled, all instances that were blocked resume their execution.

Figure 4(a) depicts the scenario where the individual instances of two group members are blocked as the barrier's condition is not yet satisfied. Once the condition is satisfied (e.g. time has passed 20:30 o'clock), the blocked instances can continue their execution. This is shown in Figure 4(b).



(a) Barrier of which the condition is not yet satisfied.

(b) Barrier with a fulfilled condition.

Fig. 4. Synchronisation: the barrier pattern.

The difference between a (normal) barrier and a *cancelling barrier* is explained by the way they treat instances that arrive at a barrier whose condition is already fulfilled (i.e. the "cancellation" criteria mentioned earlier). Individual instances of a member that arrive later at a normal barrier continue their execution without waiting. On the other hand, when a *cancelling barrier* is used, only the blocked instances will execute the remainder of the workflow after the barrier. The execution of the next instances that reach the cancelling barrier pattern are cancelled.

Remark that a cancelling barrier, by definition, has an influence on the amount of members of the group. Once the condition of the cancelling barrier pattern is fulfilled, the blocked instances of the individual members continue

their execution. From that moment on, the amount of members of the group is restricted to those that were able to continue their execution.

GroupJoin Pattern In order to terminate the execution of the group pattern both control flow and data flow must be merged. The *group join* pattern allows managing how control flow and data flow are merged. The default merging strategy used to merge the data environments of all instances is "accumulating all values for each variable", one of the merging strategies proposed in [5]. However, it is often wanted to specify another merging strategy. Therefore, the fourth optional argument to instantiate a group can be instantiated with a specific `GroupJoin` pattern.

```
Group( <description>, <variable name>, <subworkflow>, [ <group join> ] )
```

A group join pattern is instantiated with both a condition and a merging strategy. Once this condition is fulfilled, the control flow of all individual instances is merged such that the remainder of the workflow pattern after the group pattern is executed only once. Therefore, a group join pattern is, by definition, always cancelling, meaning that instances that reach the pattern after the condition is fulfilled, will not be able to continue their execution.

```
GroupJoin( <condition>, <merging strategy> )
```

In Figure 5 we show how both control flow and data flow are merged to terminate the execution of a group pattern. Note that we do not depict the group join pattern explicitly, as it is the only place the pattern is allowed.

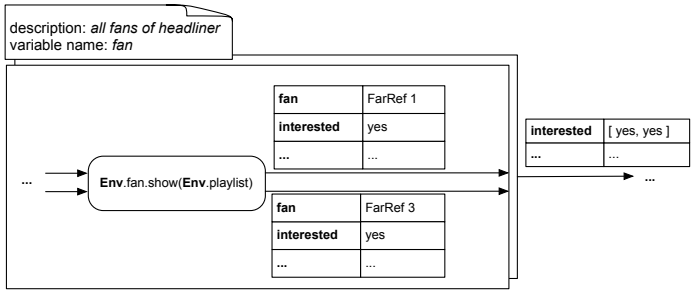


Fig. 5. Synchronisation: a group join pattern to terminate the group.

Synchronised Task In this section we present the notion of a synchronised task, a subworkflow that only needs to be executed once for the entire group at a specific moment. In essence, each member of the group executes its own instance, but there should be provisions to allow a single task to be executed *once* for several or all members of the group. Therefore, a *synchronised task* is,

by definition, always cancelling, meaning that instances that reach the pattern after the condition is fulfilled, will not be able to continue their execution.

```
SynchronisedTask( <subworkflow>, <condition>, <merging strategy> )
```

A synchronised task starts the execution of its wrapped subworkflow once a given condition is fulfilled. In Figure 6 we show the functionality of the synchronised task, that is depicted as a gray box wrapping a subworkflow (in this example, one single activity). As can be seen in the figure, both control flow and data flow of the instances must be merged before the wrapped subworkflow can be started.

During the execution of the synchronised task's subworkflow, new variable bindings can be added to the merged data environment. In this concrete example, shown in Figure 6, a new binding is added for the variable `playlist`. At the end of the synchronised task pattern, the incoming data environments of all instances are restored, and extended with those new variable bindings.

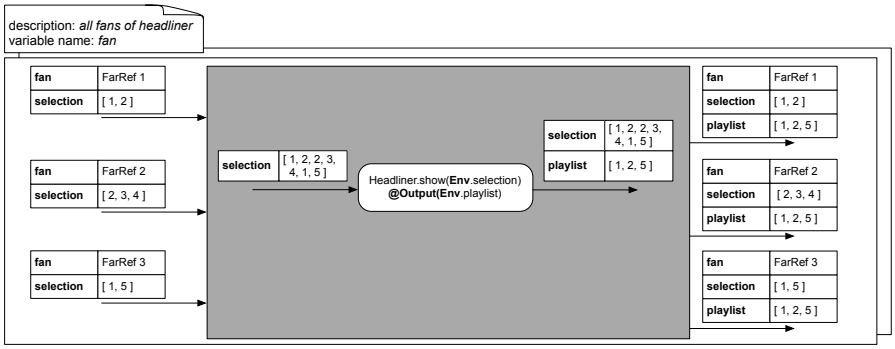


Fig. 6. Synchronised task: executing a single task synchronised.

4.3 Failure Handling [R5]

As mobile environments are liable to volatile connections, ways to detect and handle failures must be available. NOW's failure handling mechanism, described in Section 3, allows the specification of compensating actions for specific kinds of failures. In this section we present the influence of this failure handling mechanism on the novel abstractions for group orchestration we presented above. First of all, we describe *individual failure handling*, meaning the handling of failures that occurred during the individual instance of a group member. Thereafter we elaborate on how failures can be detected for the entire group.

Individual Failure Handling In order to specify compensating actions for a single individual instance of a group member, it is necessary to use the failure

pattern wrapping a subworkflow *inside* the group pattern. When using the failure pattern outside the group pattern, compensating actions might (depending on the specific type) affect the group as a whole.

```
Group( <description>, <variable name>,
       Failure( <subworkflow>, <failure descriptions>) )
```

When a failure pattern is defined inside a group pattern and wraps (part of) its subworkflow, the compensating actions only have an effect on the individual instance of that subworkflow. For instance, suppose that during the execution of a single instance a disconnection failure occurs, whose compensating action is defined as "restart the wrapped subworkflow". In that particular case, the subworkflow is restarted for that single instance (i.e. one single group member). This is in contrast with the behaviour that is accomplished when this failure pattern is defined on the outside of the group pattern.

Failure Handling for Groups It is also possible to wrap the group pattern itself with a failure pattern.

```
Failure( Group( <description>, <variable name>, <subworkflow> ),
         <failure descriptions> )
```

When the failure pattern is defined outside the group pattern, the compensating actions have an effect on the execution of the entire group. Recall the example we used earlier. When a disconnection failure occurs now, the compensating action will restart the wrapped subworkflow, in this case the group pattern. This causes all instances of the group to be cancelled and to restart the execution of the entire group.

We extended the failure handling support of NOW in order to distinguish between a failure that occurred during the execution of a group member (service), or another service, such that all failures have a *participant* variant. For instance, we make the distinction between a normal disconnection and a participant-disconnection failure. In our example scenario, we would like to specify a different compensating action for when the mobile device of single fan disconnects (for instance, drop that fan from the group), in contrast to a disconnection of the headliner (where the compensation will send a message to the crew members backstage).

The compensating actions that were provided by NOW are extended with two group-specific ones, namely *drop* and *wait-and-resume*. Both compensating actions can only be used in combination with a participant-failure (a failure that affects a group member or the communication with that member). The *drop* compensating action drops the member from the group. The second compensating action, *wait-and-resume*, can be used in combination with a participant-disconnection or a participant-not-found failure. When such a failure occurs, the place (activity) where that failure occurs is stored, such that the execution can be resumed when that specific group member (re)connects.

5 Related Work

van der Aalst [2] describes *multiple instances patterns* which wrap part of a process that needs to be instantiated multiple times. These patterns are supported by YAWL [7]. Defining a group of services can be achieved in YAWL by writing queries to retrieve the data for which the multiple instances pattern must be executed, and defining a multiple instances variable that can be used during its execution. However, this multiple instance variable does not have a reference to the service as its value, but the data resulting from the query. In YAWL it is possible to add new instances during the execution of the "multiple instances without a priori run-time knowledge" pattern. However, the synchronisation mechanisms that are supported by YAWL are rather restricted. The only synchronisation mechanism YAWL supports, is used to terminate the execution of the multiple instances pattern. There are no mechanisms like the barriers we propose provided to synchronise the execution of all instances inside the multiple instances pattern. Moreover, there is no support for the synchronised task abstraction we presented. This behaviour can be modelled in YAWL by defining a multiple instances pattern, followed by an activity or subworkflow proceeding a second multiple instances pattern. The disadvantage of this approach is that contextual information, i.e. the data for which the first multiple instances pattern was started, is discarded. The last requirement, regarding failure handling, is also not supported by YAWL. The language only has built-in support for failure handling for atomic tasks (i.e. a single activity and no subworkflow). Hence, it is not trivial to express failure handling strategies over all the activities in a group.

In the web services community, the notion of a *service group* [8] is introduced to denote a heterogeneous collection of web services that satisfy a given constraint. Such a service group only consists of fixed web services that are known beforehand (by means of a URL). Plurality encapsulation is supported as services can be added or removed from the service group, causing the service group registration to notify requestors of modifications to that service group.

Ambient References [9] enable communication with a volatile group of proximate objects by means of asynchronous message sends. Ambient references are developed as a programming language abstraction for AmbientTalk [3]. Both definition by means of an intensional description and arity decoupling are support by this language construct. Ambient references provide synchronisation mechanisms by providing observers that are triggered either when the first service has answered or when all services have responded. However, as communication with the set of services is expressed by means of an atomic message send, redefinition of group members and synchronised task abstractions do not make sense. Moreover, there are no mechanisms provided to express failure handling on an ambient reference, for instance express the action that must be performed when a service disconnects.

6 Conclusion

In this paper, we have presented the design of group orchestration patterns on top of the workflow language NOW. We introduced a novel group pattern and explained the need for advanced group synchronisation patterns. We also showed how network and service failures can be detected and handled during the orchestration of a group of services. By providing these high-level abstractions we enable orchestration with a group of services in a mobile environment where volatile connections dominate.

References

1. Mascolo, C., Capra, L., and Emmerich, W.: Mobile computing middleware. Advanced lectures on networking, pages 20–58, Springer-Verlag (2002)
2. Russell, N., ter Hofstede, A.H.M., van der Aalst, W.M.P., Mulyar, N.: Workflow control-flow patterns: A revised view. Technical report, BPMcenter.org (2006)
3. Van Cutsem, T., Mostinckx, S., Gonzalez Boix, E., Dedecker, J., De Meuter, W.: AmbientTalk: object-oriented event-driven programming in mobile ad hoc networks. Proceedings of SCCC 2007, pages 3–12
4. Mostinckx, S., Scholliers, C., Philips, E., Herzeel, C., De Meuter, W.: FactSpaces: Coordination in the Face of Disconnections. In COORDINATION 2007, pages 268–285
5. Philips, E., Van Der Straeten, R., and Jonckers, V.: NOW: A Workflow Language for Orchestration in Nomadic Networks. In COORDINATION, 2010, pages 31–45
6. Philips, E., Van Der Straeten, R., Jonckers, V.: NOW: Orchestrating Services in a Nomadic Network using a dedicated Workflow Language. Science of Computer Programming (2011), http://dx.doi.org/10.1016/j.scico.2011.10.012
7. Ter Hofstede, A.H.M.: YAWL: yet another workflow language. Information Systems, volume 30, pages 245–275 (2005)
8. Graham, S., Maguire, T., Frey, J., Nagaratnam, N., Sedukhin, I., Snelling, D., Czajkowski, K., Tuecke, S., Vambenepe, W.: Web Services Service Group - Specification (WS-ServiceGroup), Version 1.2. http://docs.oasis-open.org/wsrf/wsrf-ws_service_group-1.2-spec-os.pdf
9. Van Cutsem, T., Dedecker, J., Mostinckx, S., Gonzalez Boix, E., D'Hondt, T., De Meuter, W.: Ambient references: addressing objects in mobile networks. In OOPSLA Companion, 2006, pages 986–997
10. Van Cutsem, T., Dedecker, J., De Meuter, W.: Object-Oriented Coordination in Mobile Ad Hoc Networks. In COORDINATION, 2007, pages 231–248
11. Black, A.P., Immel, M.P.: Encapsulating Plurality. In ECOOP 93, pages 57–79
12. Guerraoui, R., Rodrigues, L.: Introduction to Reliable Distributed Programming. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.
13. Philips, E.: Website NOW. http://soft.vub.ac.be/~ephilips/NOW, 2012.
14. Luo, J., Eugster, P.T., Hubaux, J.-P.: PILOT: ProbabilistIc Lightweight grOup communication sysTem for Mobile Ad Hoc Networks. IEEE Transactions on Mobile Computing, pages 164–179, 2004
15. Peltz, C.: Web services orchestration and choreography. IEEE Computer, 36(10), pages 46–52, 2003