

**Automatic Service Composition:  
a Case for Active Networks Usability**

*Frank Matthijs  
Nico Janssens  
Pierre Verbaeten*

*Report CW 356, Jan 2003*



**Katholieke Universiteit Leuven**  
Department of Computer Science  
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

# Automatic Service Composition: a Case for Active Networks Usability

*Frank Matthijs*

*Nico Janssens*

*Pierre Verbaeten*

*Report CW 356, Jan 2003*

Department of Computer Science, K.U.Leuven

## **Abstract**

In this paper, we focus on application programmers who want to use and customize active network services. We make their job easier by simplifying the active network programming model. To this end, we decouple applications from the specific underlying active network technology, and we introduce a subsystem that automatically generates a description of a custom active networks service. The custom service description is generated from high-level application preferences, and it is automatically inserted into the application's data flow. We illustrate the system using DiANE, our active networks environment. We give concrete examples based on an active networks reliability service, showing how the reliability service is automatically customized for various types of applications.

**Keywords :** active networks, DiPS, composition.

# Automatic service composition: a case for active networks usability

Frank Matthijs, Nico Janssens and Pierre Verbaeten

DistriNet labs, Department of Computer Science, K.U.Leuven, Belgium

E-mail: Frank.Matthijs@cs.kuleuven.ac.be

## Abstract

*In this paper, we focus on application programmers who want to use and customize active network services. We make their job easier by simplifying the active network programming model. To this end, we decouple applications from the specific underlying active network technology, and we introduce a subsystem that automatically generates a description of a custom active networks service. The custom service description is generated from high-level application preferences, and it is automatically inserted into the application's data flow. We illustrate the system using DiANE, our active networks environment. We give concrete examples based on an active networks reliability service, showing how the reliability service is automatically customized for various types of applications.*

## 1 Introduction

Programmable networks are emerging as a very powerful paradigm to speed up creation and deployment of new network services. Apart from deploying the services, there is also the issue of how they are being used by applications. Depending on the type of technology, applications have a different view of the programmable network services. Open signaling provides API's that open up the routing or switching hardware and allow new services to be introduced more easily. Once these services are deployed, they are used just as any service is used in today's closed networks. Their behavior is independent of the applications that make use of the service. In other words, applications cannot customize these services. The active networks approach is an order of magnitude more flexible: it gives applications control over the network services and allows them to tailor these services to application needs. However, the cost of this flexibility is, among others, the complex programming model that active networks impose upon application developers.

In this paper, we propose automatic composition of application-specific services as a way to exploit the power of active networks while at the same time hiding their complexity from application programmers. In our model, an application provides high-level preferences from which customized network services are composed and deployed automatically.

The paper is structured as follows. The next section illustrates one of the main problems with the active networks programming model, which we call domain mismatch. We then introduce in section 3 automatic service creation and deployment as a way to avoid domain mismatch. In section 4 we illustrate the approach using DiANE, our active networks test bed, composing and deploying an application specific reliability service as an example. Before the summary, we list some known issues and future work in section 5.

## 2 Domain mismatch in active networks

Active network technology achieves its flexibility by allowing applications to insert application-specific control in the data path. Typically, imperative programs written in a specialized language are carried in network packets. Such languages can be seen as glue languages that compose node extensions together to form a customized service. These node extensions are custom pieces of software that are installed at the various nodes and that

form the basic building blocks for more elaborate services. Examples of systems that follow the active network model are ANTS [1] (capsules), PLAN [2] (deliberately restricted glue language), Smart Packets (network management) [3], SANE (authentication) [4], SafetyNet (system integrity) [5], M0 (messengers) [6].

The fact that the network can be programmed to offer application-specific services is a very powerful concept and it enables a whole range of new possibilities. However, writing programs in these languages is not trivial and it requires thorough knowledge of the active network infrastructure, the technical details of the node extensions, and how they are to be used together to create a new service.

The key problem here is that typical application programmers are not network specialists. They know the application domain very well, but in general they do not know how to develop a network service. We call this problem *domain mismatch*: application programmers, who wish to customize the way their applications are executed, are suddenly confronted with a domain (in this case, active networks) that is completely different from the familiar application domain. Developers obviously find it difficult to express application customizations in terms of the unfamiliar domain.

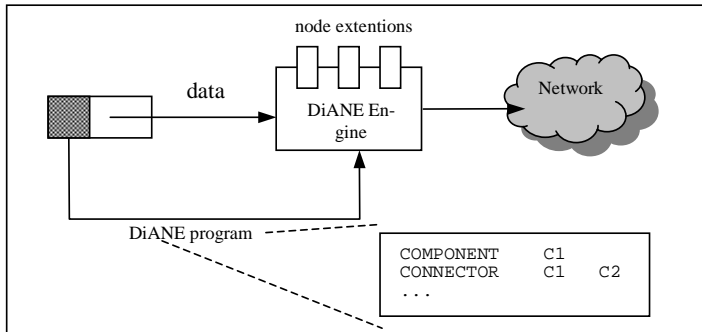
In order to analyze the problem, and as a first step towards a solution, we introduce a new role in active networks application development: apart from the *application programmer*, who develops applications that run on a network, and the *network programmer* who creates network services or node extensions using programmable network API's, we introduce the *application deployer*, who makes sure a specific application runs optimally on a given network. Application deployment on a network with fixed services is easy: nothing has to be configured; the application simply runs (or it doesn't). On a network with customizable services, application deployment involves configuring the network services to match the application's preferences. In an active networks context, we can customize the services at run-time, even on a per-packet basis (or more realistically, on a per-session basis). We will focus on the case where the application deployer composes services using programs written in the active network's glue language, and inserted into the application's data flow.

The question now is who will play the role of application deployer. An obvious choice is to have the application developer do this. However, as explained above, this solution suffers from domain mismatch. The network programmer is another obvious choice, because that is where the knowledge about the inner workings of the network is. However, without knowledge of the application, or at least of some characteristics of the application, the network programmer cannot perform this task. The solution we propose in this paper is to automate the customization of network services based on high-level information provided by the application developer. To cope with the domain mismatch problem, this high-level information is not expressed using technical networking terminology, but rather uses keywords that are either generic or close to the applications' problem domain.

## 3 Automatic service composition to the rescue

### 3.1 DiANE

Before explaining automatic service composition, we first give an overview of our active networks setup. It is based on DiANE (Distrinet Active Networks Environment), our execution environment for active packets. Figure 1 illustrates the way the engine works. Network packets processed by a DiANE engine contain a program that defines the way in which the packets are to be processed. This program describes a composition of node extensions as a number of components connected by connectors.



**Figure 1: working principle of a DiANE engine**

At this level, the program defines a data flow in a way similar to Netscript [7]. However, a DiANE program is more expressive: it does not only specify the components (node extensions) that are to be used and the connections between them, it also specifies the type of connections. Each connection type is represented by a specific connector. The separation into components and connectors is a fundamental concept from DiPS (Distrinet Protocol Stack) [8], our framework for building network subsystems, which we used to build DiANE. It provides good separation of concerns between functional aspects (implemented in components) and non-functional aspects (implemented in connectors). For example, the concurrency model of a networking subsystem is completely defined by the type of the connectors that are used. The components can be reused regardless of the concurrency model. Since a DiANE program defines both the components and the connectors a service is built from, it can express not only the functionality of the service, but also aspects such as its concurrency model.

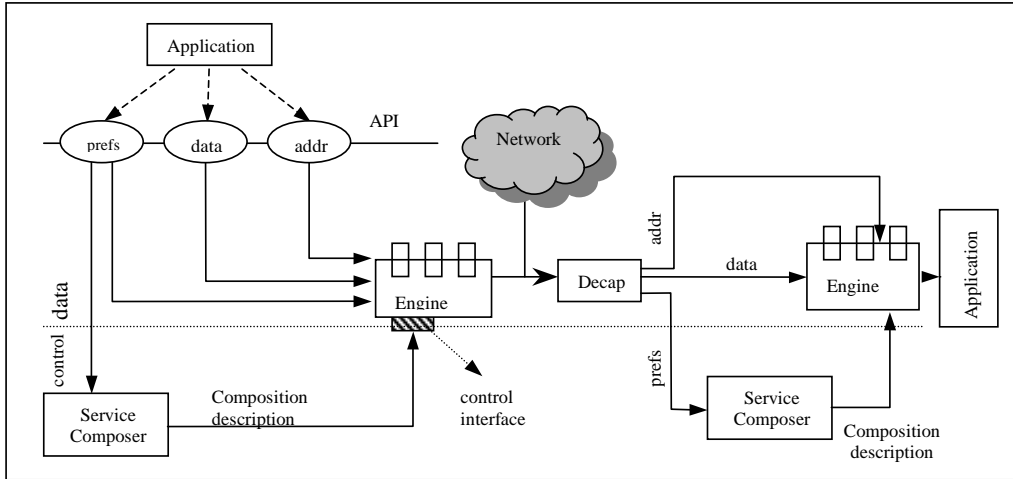
The DiANE architecture contains an interpreter on each node (the DiANE engine) that dynamically creates the composition described in incoming packets or packets originating from that node, and then processes the packets using that composition. In this way, users can compose their own network service and send a description of the exact composition along with the packets that are to be processed by the service.

At intermediate nodes and at the destination node, the same process is repeated: an incoming packet is processed by the components that form the composition described in the packet. Note that in this way, the application-defined service is also automatically deployed on the network: packets contain a description of the service they want, and this service is built dynamically on all intermediate nodes wherever it is needed. Additional packets that require the same service are processed immediately by the previously built composition; there is no need to rebuild the service over and over again for each incoming packet. Removal of services can be performed according to different policies. For example, if the service does not maintain state, it can be safely removed at any time since the next packet that needs that service still contains a description of its composition.

In its basic form, our active network architecture does not in any way solve the domain mismatch problem: the application deployer still has to produce the service composition description that is to be inserted in the packets. Therefore, we introduce the service composition subsystem that will create this description automatically, based on information that is coming from the application developer.

### 3.2 DiANE's Service Composition Subsystem

This subsystem is responsible for creating the service composition description that is used by DiANE's packet processing engines. The idea is that each service can have a companion *service composition module* plugged into this subsystem. The companion module knows how the service is built up from components. In addition, if the service is customizable, the companion module knows how all the variants are built. For example, a reliabil-



**Figure 2: General architecture**

ity service might be customizable so that it uses either positive acknowledgements, selective acknowledgements or negative acknowledgements. In that case, the companion module knows how to create the reliability service for each of the three possibilities.

Continuing our example, a typical usage scenario could be that an application indicates that it wants to make use of the reliability service for some session, and that it wants positive acknowledgements to be used. The service composition subsystem translates this request into a description of a reliability service with positive acknowledgements, and it inserts this description into the application's packets that are sent in that session. As explained in the previous section, the DiANE engines on the sending node, all intermediate nodes and the receiving node will dynamically build the requested (custom) service from this description and use it for processing the packets in that session.

Figure 2 illustrates the general architecture. In this case, the application's preferences are sent integrally with the data, to each intermediate node and to the destination. At each node, this program is processed by the local DiANE engine and translated into a service composition description. The advantage is that the resulting service does not have to be identical on all nodes. Each node can instantiate its own version, for example depending on whether the node is an intermediate node or a destination node.

A simplified setup is shown in figure 3. Here, the composition is determined only once and its description, in the form of a DiANE program, is sent with the packets. This setup deploys the same local service part on all nodes. In this case, the service composition module and the service composition subsystem need only be present on the sending node, which makes the system easier to manage.

Whichever configuration is chosen, the service composition subsystem can generate service descriptions that do not only take into account the application preferences, but also local circumstances. For example, it might generate a slightly different service composition description when it observes that local conditions, such as an erratic network connection, might influence the behaviour of the service. Obviously, the configuration from figure 2 is more flexible in this regard.

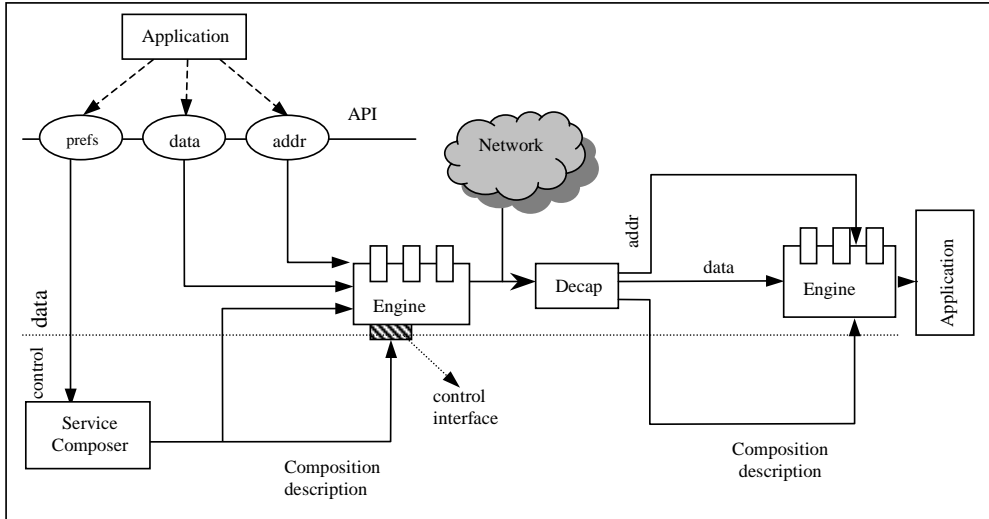


Figure 3: Simplified architecture

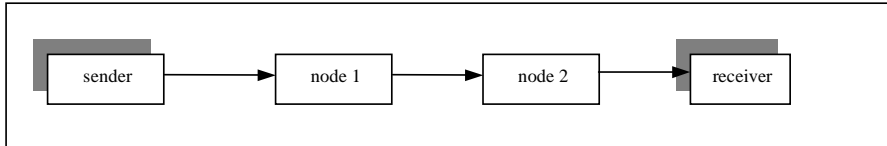
### 3.3 Decoupling and abstraction

The automatic service composition subsystem described in the previous section is already a large improvement for application developers: they do not need to know how a specific service is built; they just indicate that they want to make use of this service and provide some customization options. The service composition subsystem takes care of the rest. However, typical application developers do not really know, for example, in what circumstances positive, negative or selective acknowledgements are appropriate. On the other hand, they do know whether delays (such as delays caused by large retransmission time-outs in combination with positive acknowledgements) are acceptable or not in their application. Therefore, the companion modules in DiANE make use of *high-level information* that is expressed either in general terms, or in terms of the application domain. This is the kind of information that application developers can typically provide. In the case of our example, the application simply indicates whether it is delay-sensitive.

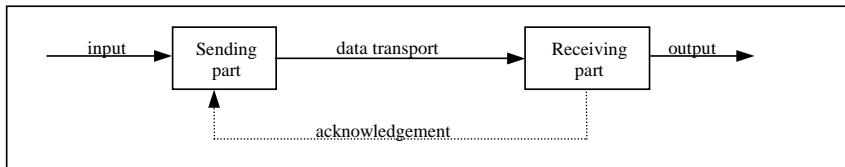
By raising the abstraction level of the information that applications have to provide, we effectively solve the domain mismatch problem. An additional advantage is that it reduces coupling between what an application has to provide and the specific underlying technology. For example, applications do not have to be rewritten when a different active network technology is chosen; only the specific service composition modules have to be (re)developed.

In terms of the roles we introduced earlier, there is no strong dependency anymore between the application developer and the network programmer. Each can perform his or her task independently and applications and network service building blocks can be individually reused or upgraded. In between sits the service composition subsystem and its service composition modules that play the role of (automatic) application deployer. They bind an application to the underlying networking technology, at run-time.

A service composition module is typically developed by network programmers. It captures their expertise in building a network service from basic building blocks. This is actually a very interesting point: it shifts the creation (or composition) of network services back from application developers to network programmers, where it probably belongs. This shift also reintroduces a market for services. Developers can create simple network services that are only minimally customizable, or they can market a very sophisticated service with a lot of customization options and a very intelligent service composition module. They can even market the service composition modules separately.



**Figure 4: basic configuration**



**Figure 5: reliability components**

An obvious disadvantage of raising the abstraction level and decoupling applications from the underlying network is that more complex infrastructure support is required, such as the DiANE service composition subsystem and companion modules for the different services. However, we feel that this kind of support is essential for the wide-spread acceptance of active networks (or any complex environment for that matter), and also for making sure that their potential is used to its fullest. After all, an environment is nothing if no applications exist to take advantage of its features.

## 4 Case study: an application-specific reliability service

In this section we consider a customizable reliability service and we illustrate concretely how applications can make use of this service and how they can customize it. The complete setup makes use of DiANE.

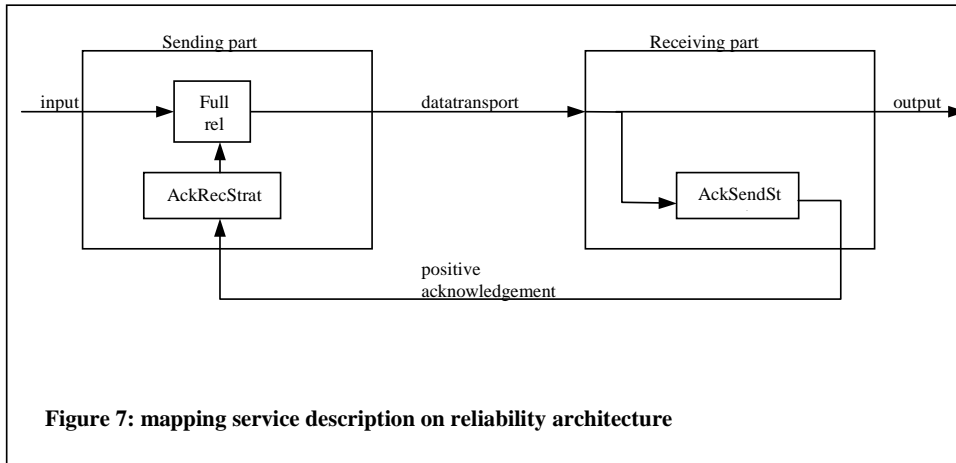
The basic configuration is illustrated in figure 4. We have four nodes, all part of an active network. Every node runs the DiANE packet-processing engine. The leftmost node (labeled “sender”) runs an application that wants to send data; the rightmost node (labeled “receiver”) runs an application that receives the data. The intermediate nodes are labeled “node 1” and “node 2”. We assume that all nodes have basic UDP capabilities. Depending on the specific case, some nodes also have the service composition subsystem available. In all cases, the sender node has the service composition subsystem installed with the companion module of the reliability service.

The reliability service itself, like any reliability mechanism, consists of two parts in each direction, on each node where it is installed, see figure 5. The sending part will resend the delivered data until it receives an acknowledgement from the receiving part, in order to guarantee the required degree of reliability.

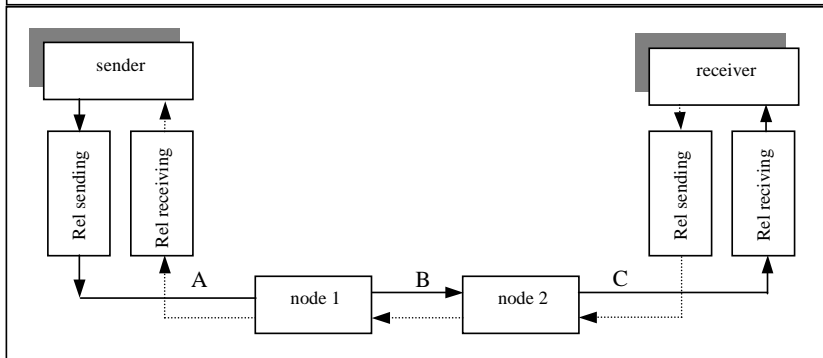
Our reliability service is designed to accommodate a wide range of applications. Indeed, not all applications have the same requirements for reliability. We will illustrate some cases in the remainder of this section, introducing the customization options as we go.

### 4.1 Full reliability

Here we consider cases such as file transfer applications, which need full reliability. These applications cannot rely on UDP and will therefore use our reliability service. They simply indicate that they want fully reliable communication, and the service composition subsystem takes care of the rest. In this case, it will generate, through the reliability companion module, a description of a reliability service that offers full reliability.



**Figure 7: mapping service description on reliability architecture**



**Figure 8: reliability components at sender and receiver**

Figure 6 shows the generated service description for this case. The figure contains parts that are not really relevant to our discussion (such as the connectors that glue the components together). We only explain the most important items. Basically, the description selects node extensions for *full reliability* and for *positive acknowledgements*. When we look to the service description in more detail, the global structure can be mapped to the architecture described in figure 5. This mapping is illustrated in figure 7. The down going path (sending part) selects a `FullReliability` node extension, which is responsible for the buffering and resending of each packet that comes by. The part of the service description for the up going path contains the positive acknowledgement node extensions in order to send and analyze acknowledgement-supporting packets, resp. `AckSendingStrategy` and `AckReceivingStrategy`. Reflection points route the data between node extensions depending on its type: acknowledgements are routed differently from normal data.

Apart from the service composition itself, the reliability companion module has several choices about the deployment of the service. The most obvious is to deploy the reliability components only at the sending and receiving node, as illustrated in figure 8. Intermediate nodes simply route the data at the network level. This resembles most closely standard reliability protocols such as TCP.

However, this is not the only possibility. Consider there is an unreliable wireless link (link C) at the end of the communication path. Due to the packet loss on this link, the sender will initiate a high number of retransmissions, all of which have to travel over link A and B as well. A better solution might be to handle the problem locally. To achieve this, the reliability components can also be instantiated at the intermediate nodes (see figure 9). Now every link is responsible for the required reliable communication. When a packet is dropped at link C, it will only be resent over C, instead of starting from the sender over link A and B.

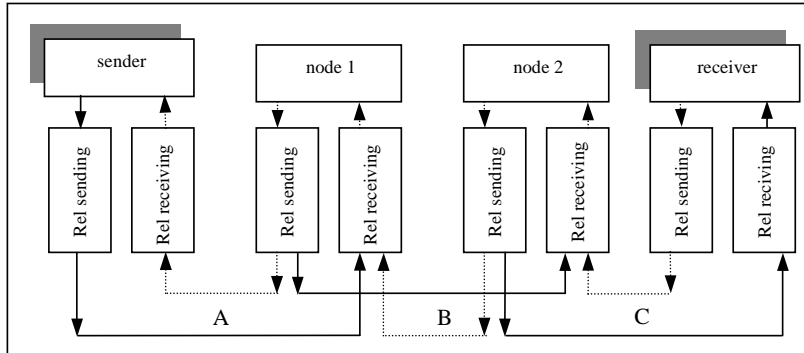
```

; Downgoing Path
; -----
;
; Components
;
COMPONENT dips.protocol.reliability.downpath.component.ControllerComponent contrcomp
COMPONENT dips.protocol.reliability.downpath.component.FullReliability fullrelstrat
COMPONENT dips.protocol.reliability.ReliabilityHeaderConstructor headerconstructor
;
; UpperEntryPoint
;
UPPERENTRYPOINT contrcomp
;
; Connections
;
CONNECTOR dips.repository.connector.SimplePipeConnector cd1 contrcomp fullrelstrat
CONNECTOR dips.repository.connector.SimplePipeConnector cd2 fullrelstrat headerconstructor
CONNECTOR dips.repository.connector.SimplePipeConnector cd3 headerconstructor downdelivery
;
; Upgoing Path
; -----
;
; Components
;
COMPONENT dips.protocol.reliability.ReliabilityHeaderParser headerparser
COMPONENT dips.protocol.reliability.uppath.component.AckSendingStrategy acksendstrat
COMPONENT dips.protocol.reliability.uppath.component.AckReceivingStrategy ackrecstrat
COMPONENT dips.protocol.reliability.uppath.component.NextSequenceFilter nextsequence
;
; LowerEntryPoint
;
LOWERENTRYPOINT headerparser
;
; ReflectionPoint
;
REFLECTIONPOINT dips.protocol.reliability.routing.PacketTypeReflector packettyperout
REFLECTIONPOINT dips.protocol.reliability.routing.PacketTypeReflector ackoutputrout
;
; Configuration of reflection points
;
ROUTPUTPUT ack packettyperout ackrecstrat
ROUTPUTPUT data packettyperout nextsequence
ROUTPUTPUT ack ackoutputrout headerconstructor
ROUTPUTPUT data ackoutputrout updelivery
;
; Connections
;
CONNECTOR dips.repository.connector.SimplePipeConnector cu1 headerparser packettyperout
CONNECTOR dips.repository.connector.SimplePipeConnector cu2 nextsequence acksendstrat
CONNECTOR dips.repository.connector.SimplePipeConnector cu3 acksendstrat ackoutputrout

```

**Figure 6: generated composition description for full reliability**

DiANE's service composition subsystem can deploy the service either way. For deploying the same service composition on all nodes, the simplified setup (see figure 3) suffices; for a deployment with different compositions on each node, the general setup of figure 2 should be used, with different companion modules on the intermediate nodes, for example.



**Figure 9: reliability components in each node**

The second case performs better than the first under some circumstances. Still, the application does not need to be modified. This illustrates one of the advantages of our decoupling: the service companion modules can choose to instantiate an optimized version of the service without the application having to care about this technical aspect.

## 4.2 Delay requirements

Referring back to an earlier example, the reliability companion module is also able to interpret delay requirements of the application. The application simply indicates whether it is sensitive to delays on the connection or not. When an application indicates high delay sensitivity, the reliability companion module tries to keep the retransmission time-outs as small as possible. To that end, it includes dynamic timer components in the reliability service. These components frequently estimate the actual round-trip times and update the retransmission values accordingly. Figure 10 shows the generated composition description for this case. The composition contains the same components from figure 6, but adds dynamic timer support by means of the `TimeStampAttacher` and the `RoundTripTimeCalculator`, resp. on the down and upgoing path.

## 4.3 Jitter control

Instead of needing full or no reliability, there are applications that require some, but not full reliability. Consider an industrial robot application, where packets with new position-coordinates for the robot arm are sent at a certain frequency, say 100 Hz. At the end of each sending period, 10 ms, a new position will be generated, which will render the current position packet obsolete. At that moment, there is no point in continuously resending the current position packet if it hasn't been correctly received yet.

Applications such as our robot application and most multimedia applications will perform better when the reliability service can cope with this situation. The major difference between full reliability and its multimedia counterpart is the degree of reliability. When using full reliability, every packet must be acknowledged before it is dropped at the sender-side. In case of multimedia reliability, the packet will also be dropped when a new one has to be delivered. An important advantage of dropping old packets is that it reduces *jitter* (the variation on the delay of packets received at the destination) [9].

Our reliability composition module supports this kind of applications. They indicate to the composition subsystem that they are jitter-sensitive, and the composition module automatically incorporates the necessary components. In this case, it replaces the `FullReliability` component from figure 6 by a `MultimediaReliability` component.

```

; Downgoing Path
; -----
;
; Components
;
COMPONENT dips.protocol.reliability.downpath.component.ControllerComponent contrcomp
COMPONENT dips.protocol.reliability.downpath.component.FullReliability fullrelstrat
COMPONENT dips.protocol.reliability.repository.TimestampAttacher timeattach
COMPONENT dips.protocol.reliability.ReliabilityHeaderConstructor headerconstructor
;
;
; UpperEntryPoint
;
UPPERENTRYPOINT      contrcomp
;
;
; Connections
;
CONNECTOR dips.repository.connector.SimplePipeConnector cd1 contrcomp fullrelstrat
CONNECTOR dips.repository.connector.SimplePipeConnector cd2 fullrelstrat timeattach
CONNECTOR dips.repository.connector.SimplePipeConnector cd3 timeattach headerconstructor
CONNECTOR dips.repository.connector.SimplePipeConnector cd4 headerconstructor downdelivery
;
;
; Upgoing Path
; -----
;
; Components
;
COMPONENT dips.protocol.reliability.ReliabilityHeaderParser headerparser
COMPONENT dips.protocol.reliability.uppath.component.AckSendingStrategy acksendstrat
COMPONENT dips.protocol.reliability.repository.RoundTripTimeCalculator ackrttcalculator
COMPONENT dips.protocol.reliability.uppath.component.AckReceivingStrategy ackrecstrat
COMPONENT dips.protocol.reliability.uppath.component.NextSequenceFilter nextsequence
;
;
; LowerEntryPoint
;
LOWERENTRYPOINT      headerparser
;
;
; ReflectionPoint
;
REFLECTIONPOINT dips.protocol.reliability.routing.PacketTypeReflector packettyperout
REFLECTIONPOINT dips.protocol.reliability.routing.PacketTypeReflector ackoutputrout
;
;
; Configuration of reflection points
;
ROUTPUTPUT      ack      packettyperout  ackrttcalculator
ROUTPUTPUT      data     packettyperout  nextsequence
ROUTPUTPUT      ack      ackoutputrout  headerconstructor
ROUTPUTPUT      data     ackoutputrout  updelivery
;
;
; Connections
;
CONNECTOR dips.repository.connector.SimplePipeConnector cu1 headerparser packettyperout
CONNECTOR dips.repository.connector.SimplePipeConnector cu2 ackrttcalculator ackrecstrat
CONNECTOR dips.repository.connector.SimplePipeConnector cu3 nextsequence acksendstrat
CONNECTOR dips.repository.connector.SimplePipeConnector cu4 acksendstrat ackoutputrout

```

**Figure 10: generated service description for full reliability and delay requirements**

However, very often an application wants to keep some control over the degree of reliability that is achieved. Therefore, we allow applications to indicate the degree of reliability they need. In general, more reliable communication leads to more jitter as well. The application has to decide what degree of reliability it requires or what degree of jitter it is able to accept. To be able to offer certain reliability guarantees, the reliability composition module selects a dynamic multiple sending component in addition to the `MultimediaReliability` component. Instead of sending a packet just once, several copies will be delivered depending on the measured drop rate of the involved communication channel. By tuning the number of packet copies that are sent out, the reliability service can guarantee an average degree of reliability.

In its most sophisticated form, our reliability service measures the drop rate on the connection and adjusts the service parameters automatically. As for the deployment of this service, we again have the option of deploying

the same composition on every node, or creating a specialized version on each node. Similar to the retransmission case where bad links are taken care of locally, drop rates can be estimated and parameters tuned on each local link individually when we deploy the full service on each node. This leads to less waste of bandwidth at the expense of a more complex system.

## 4.4 Combined preferences

Of course, several application preferences can be combined. For example, for an application that indicates that it is jitter-sensitive, delay-sensitive, and requires 70% reliability on average, the composition module for the DiANE reliability service generates a composition description that includes:

- Dynamic timer support to provide delay restriction.
- Multimedia reliability component to provide jitter restriction.
- Drop rate estimation and dynamic multiple sending support to provide the required 70 % of reliability while keeping the jitter as low as possible.

## 5 Issues and future work

There are a number of important issues that we do not currently address in our active networks environment. One of them is security. In a highly dynamic environment such as programmable networks, there is always the issue of who is allowed to install what and where (authorization), and how we can prevent malicious users, modules, etc. from compromising the system (safety). In this paper, we assume that all the necessary node extensions are already fully verified and safely installed. Still, the question about what applications are allowed to compose what services, remains. With a programming model where applications can send any kind of active program with their data, authorization and safety are an important concern that is difficult to address. Although we do not explicitly address this issue, the model with an explicit service composition subsystem makes it a lot easier to implement the necessary security mechanisms. For example, the safety issue can be solved by verifying the composition modules once and installing them in a safe way. After that, applications making use of the service can never directly compromise the integrity of the system: all they can do is provide their preferences, the actual composition is generated by a verified and therefore trusted component in the system. Authorization might still be necessary, but the problem is reduced to deciding what application preferences to take into account.

Regarding our deployment model, it is currently based on the assumption that all the necessary node extensions are installed at the various nodes of the network. It is currently not clear whether this is an advantage or a disadvantage. On the one hand, it might simplify things if node extensions could be downloaded automatically from some repository. On the other hand, local administrators will probably not trust extensions that originate from outside their administrative domain, so there is a need for at least one repository per domain, where extensions can be verified by the local administrators, and a transport medium that guarantees integrity. It remains to be seen what model will be acceptable for system administrators. Until then, we simply assume that everything is installed when we need it.

One problem with our approach is that it currently handles one service at a time. Also, the application has to refer explicitly to the specific service it wants to use. However, this has the same disadvantages as directly specifying the node extensions that are to be used. For example, when a new service becomes available, applications have to be aware of this so that they can refer to the new service if necessary. However, we can apply our model to service selection as well: applications simply indicate high-level preferences and the system will figure out what services it will use and how the services will be customized. This is especially important when there are multiple services. When many services exist, it becomes increasingly difficult for application developers to compose them together. This is actually the same story for multiple services than what we discuss in this

paper for a single service. We are currently working on a system that selects, based on a number of high-level application preferences, a suitable composition of services.

Finally, the mechanism by which applications express their preferences is currently quite ad hoc. We are working on a mechanism whereby a service can publish a vocabulary of customization keywords that it can understand. Applications then use that vocabulary. The mechanism is independent of a particular service.

## 6 Summary

In this paper we address the usability problem of active networks, by proposing automatic service composition as an intermediate between applications on the one hand and active networks technology on the other hand. We present DiANE, our active networks environment, which includes a service composition subsystem that is responsible for this automatic composition, and which thus binds a specific application to a specific underlying network. Using a customizable reliability service as an example, we illustrate a number of applications with different requirements, and the mapping from those requirements onto the underlying network. The examples illustrate the key points of this paper:

- Determining the correct composition of a reasonably complex service is not straightforward. Components might need to be composed in a certain order, some compositions do not make sense (such as dynamic multiple sending support without drop rate estimation, in an attempt to ensure a required percentage of reliability), etc. In general, each composition has its own characteristics, and these characteristics can only be known and fully understood by specialists in the corresponding domain.
- Application developers do not know how certain details of a particular service, such as it using large retransmission time-outs in combination with positive acknowledgements, influence the observed characteristics (or quality) of the requested service. They do know what kind of service they need, but this is expressed in high-level terms.
- In DiANE, the expert knowledge about service compositions is provided by companion modules that plug into the service composition subsystem. Each module has the ability to map high-level application preferences onto a suitable composition of the service it is associated with. These modules can be simple if-statements covering a number of customization choices, or they might implement a complete expert system. As far as the application is concerned (at least from a design perspective), this does not really matter. It simply influences how well the service can be tailored to the application, and thus how well the service will perform for that particular application.
- A full service consists of a number of components that are the basic building blocks of the service, and a companion service composition module. The sophistication level of the composition modules determines the degree of customization of the service, as well as the level of optimization that is performed. The modules can be marketed separately.

As always when things are generated automatically, the system we propose is not the best solution in all cases. Some applications might want to have full control over the network services. We strongly believe, however, that most application developers need extra support in deploying their applications in a complex environment. Active networks certainly qualify as complex environments, and they suffer as such from usability problems. Therefore, we believe that support such as what we describe in this paper is a crucial precondition for the widespread acceptance of active networks.

## 7 References

- [1] David J. Wetherall, John Guttag, and David L. Tennenhouse. "ANTS: A toolkit for building and dynamically deploying network protocols." In *Proceedings of the 1998 IEEE Conference on Open Architectures and Network Programming (OPENARCH '98)*, April 1998.
- [2] Michael Hicks, Pankaj Kakkar, Jonathan T. Moore, Carl A. Gunter, and Scott Nettles. "PLAN: A packet language for active networks." In *Proceedings of the 1998 ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, September 1998.
- [3] B. Schwartz, A. Jackson, T. Strayer, W. Zhou, R. Rockwell, and C. Partridge. "Smart packets for active networks." In *Proceedings of the 1999 IEEE 2nd Conference on Open Architectures and Network Programming (OPENARCH'99)*, March 1999.
- [4] D. Scott Alexander, William A. Arbaugh, Angelos D. Keromytis, and Jonathan M. Smith. "A secure active network environment architecture: Realization in SwitchWare." In *IEEE Network*, 12(3), 1998.
- [5] Ian Wakeman, Alan Jeffrey, Rory Graves, and Tim Owen. "Designing a programming language for active networks." <http://www.cogs.susx.ac.uk/projects/safetynet/papers/isdn.ps.gz>, June 1998.
- [6] Christian F. Tschudin. "The messenger environment M; a condensed description." In *J. Vitek and C. Tschudin, editors, Mobile object systems; towards the programmable Internet, LNCS 1222*, April 1997.
- [7] Y. Yemini and S. da Silva. "Towards Programmable Networks." *IFIP/IEEE International Workshop on Distributed Systems: Operations and Management*, L'Aquila, Italy, October, 1996.
- [8] F. Matthijs, "Component framework technology for protocol stacks." Ph.D. thesis, Department of Computer Science, K.U.Leuven, Leuven, Belgium, 1999.
- [9] N. Janssens. "Optimalisatie van een gedistribueerde robotsturing." Master's thesis, Department of Computer Science, K.U.Leuven, Leuven, Belgium, 2000.