

Collection of Network Information in Active Networks

Yuhong Li, Lars Wolf
Institute of Telematics, University of Karlsruhe
76128 Karlsruhe, Germany
{li, wolf}@telematik.informatik.uni-karlsruhe.de

Abstract

Collection of information about the state of the network is very important for applications both in active networks and in traditional passive networks. However, the existing methods for the collection and prediction of network state cannot meet the needs of complex and intelligent applications. Active networks provide an environment that supports the injection and execution of application-specified codes in the network nodes, and consequently suggest possibilities for the solving of such a problem. In this article we present a new method for the collection of network state information by describing how active network techniques can be used to collect this information more freely and in a user-specific way. We give also an implementation of this approach to collect network resource information and evaluate it by measurements. By using this method, users can specify both the content and the format of the network resource information that they want to acquire as well as nodes at which they want to gather these information.

1. Introduction

In recent years active networks attracted a lot of attention because of their capability of enabling new user-specific services into networks promptly. More and more applications are being developed aiming to accomplishing protocol upgrading/revision, solving specific hard network problems, such as network management, congestion control and multicasting etc, as well as fulfilling user-specific data processing in network nodes. As active networks introduce user-specific computation into the network nodes, applications in active networks use the networks and the resources in network nodes more aggressively than simple applications in traditional passive networks, and therefore their performance becomes more sensitive to the network conditions, such as available resources in the network. Typically, active networks still belong to the class of “best-effort” networks, and the performance of applications using them cannot be guaranteed, which is a well-known drawback of “best-effort” networks, even though there will be more and more resources available inside the networks with the progress in technologies. In order to acquire better performances, these applications need sufficient information about the resource state within the network. But collecting network state information is not easy in active networks, sometimes even very difficult:

applications may need a wide variety of information about the network, ranging from static network topology, dynamic bandwidth estimations on a variety of time scales, to latency information in different network sections. In addition, different applications may lay emphasis on different resources. This presents a challenge of how to formulate a rich set of network information in a not too complex way. Besides this, network conditions can change very quickly, hence how to acquire the most recent information is a problem. Normally application level connections between network nodes share physical links with other connections, therefore sometimes one application wants to know which and how many resources are available to itself. Furthermore, active networks may use diverse technologies as their infrastructure and can be very large, this makes it difficult for an application to calculate the resources available to itself and to acquire resource state information along network nodes. From another point of view, it is clear that the current traditional networks cannot be replaced by the active networks completely due to economical and technological reasons. Active networks and traditional passive networks will co-exist for a long time. Hence in some cases it is also necessary for an active application to locate active nodes where the application-specific code can be executed. Generally speaking, it is difficult yet necessary both to acquire and to represent the network information properly.

In fact the collection of network information is not only needed in active networks. With the development of complex, intelligent and network-aware applications [4][8] in traditional networks, some efforts have been tried to find better methods for collecting information about the state of the networks. The simplest method that has been used to acquire network information is to use of the implicit network feedback, e.g. to interpret packet loss as a sign of congestion. But this method can give only a little information. Another method is probing the network by using benchmarks, a small set of representative applications. Yet this introduces overhead for both the application and the network, and furthermore, the mapping from benchmark performance to application performance is also a challenge. The well-known traceroute program enables a user at a host to get a list of all the routers that the packets from this host to a specified destination pass by with the elapsing time to reach them. However, it can only retrieve the hostname and the delay along a path. SNMP (Simple Network Management Protocol)[3] allows users to ask the network information

through an interactive protocol between users and networks, but not all of the applications have the authority to get all the information that they want. Hence some complicated systems that focus on collecting network information, such as Globus[6], Prophet[16], NWS[20] etc. have been designed. But more or less they still could not meet the need of complex and intelligent applications. We will consider these methods in more detail in section 4.

Obviously, the above methods for collecting network state information in traditional networks cannot be directly used in active networks for the sake of the distinctive characteristics of applications in active networks. The goal of this paper is to propose a method that can better solve the problem of collecting state information about intermediate network nodes to better satisfy the need of the users in active networks. Our method allows users to ask for information about resources at any active network node along the way to the destination. Besides this, the network information can be presented in any format, i.e. the user controls the way in which the network information is presented. This omits the translation of the information between network and user and between different network protocol layers. In short, through our approach, the resource capabilities of any user-specified active node can be acquired and presented in a user-specific way. This lays also a foundation for the research of resource management in active networks.

The remainder of the paper is organized as follows. Section 2 gives a brief introduction to active networks and presents an active network node architecture that we used to collect the network information. In Section 3 we present our approach for the collection of network node information, including its function, the capsule format, implementation method and some results. We contrast our approach with related work in Section 4, and make a conclusion in Section 5.

2. An Active Network Node Architecture

2.1 Active Networks

The concept of active networks emerges from discussions within the broad DARPA research community in 1994 and 1995, and one of the first descriptions is given in [15]. Its most important characteristic is to allow customized computations in the network nodes. This requires the network node to be able to provide an execution environment (EE) for the execution of the customized computation. Correspondingly the operating system in the network node needs to do some adjustment in order to be able to cooperate with the EE. Unlike in the traditional layered network, where by standardizing the syntax and semantics of the data frames of each layer,

users or applications in different layers at the network and end-system can communicate with their peers, in active networks, in order to realize the interoperability between network and user, the interface between the EE and user applications is standardized.

The above characteristic of active networks brings some overwhelming advantages compared to conventional networks: simple integration of new technologies and standards into the shared network infrastructure; better performance that benefits from overcoming the redundant operations at several protocol layers; and the ease of accommodating new services in the existing architectural model[11]. Active networks can be potentially very adaptable and extensible, while they can still provide core services, and backward compatibility.

As mentioned before, in order to realize the customized computation in the network, the network node must be able to provide an EE to the computation and the supporting OS. Hence, in order to make the network active, such an EE and OS must be implemented. Over the last years there has been considerable effort at many universities and institutes to provide an efficient, flexible and safe architecture of active nodes. Examples are ANTS[18] and PAN[10] in MIT, SwitchWare[1] in the University of Pennsylvania, and PANTS[5] in the University of Sydney etc. Our approach makes use of ANTS. Hence we describe it in some more detail in the following.

2.2 ANTS

ANTS is a toolkit that allows experimentation with active network protocols[17][18][19]. An ANTS based network can be seen as a conventional wide area network in which some of the IP routers have been replaced by active nodes which possess the ANTS runtime environment. ANTS has three main components. Capsules, taking the place of traditional packets, contain application related data and describe the processing they require within the network. Active nodes, replacing the selected routers and end nodes, perform this processing and maintain the state associated with network services in a way that keeps services from damaging the network or interfering with each other. The code distribution system ensures that the processing descriptions are automatically transferred to the nodes that require them. In order to depict our method of collecting network information, we enumerate in the following some characteristics of ANTS that we used in our implementation.

Capsule

Capsules are the data units passing through the networks in ANTS. One capsule carries a reference to the method used to process the capsule at each active node, and the

method may be already cached in the active node or carried as the payload associated with the capsule.

Just like a normal packet, the format of a capsule consists of two parts: header and payload. The header involves an ANEP header[2] which consists of mainly a destination and a source option field. In ANTS, some related capsules are organized to a code group, and a collection of related code groups are treated as a protocol, which is a single unit of protection by active nodes: capsules that belong to the same protocol can access the same resources. An identifier for such a protocol and a particular capsule type within that protocol are carried in the payload of each capsule. The remainder of the payload carries code and/or data, depending on the type of the capsule.

Code Distribution

In an active network, a mechanism is needed for propagating program definitions to the nodes where they are needed. ANTS adopts an on-demand load procedure to transport code that will be used to process capsules in the network nodes. The program code loading procedure is illustrated in figure 1.

When a capsule arrives at a node, the node will check whether the method for the processing of this capsule is already available in this node according to the type identifier carried in the capsule. If yes, the corresponding method for the processing of this capsule will be invoked and the data in the capsule will be processed using this method.

If the method for the processing of this type of capsule is not yet available, the content of the capsule will be first saved in a waiting queue, and the node will generate a load request message in a form of LoadReq capsule and send it back to the previous node from which the node has first received the capsule. When the previous node receives a LoadReq capsule, it will send the required code according to the type identifier carried in the LoadReq capsule in one or several LoadResp capsules, according to the size of the code and the maximum length limit of one capsule. When the node receives a LoadResp capsule, the methods carried in this capsule will be cached in the node. Finally, the required code is available and the waiting capsules can be processed. If the desired responses are not received in a specified time period, the queued capsules are discarded without further action.

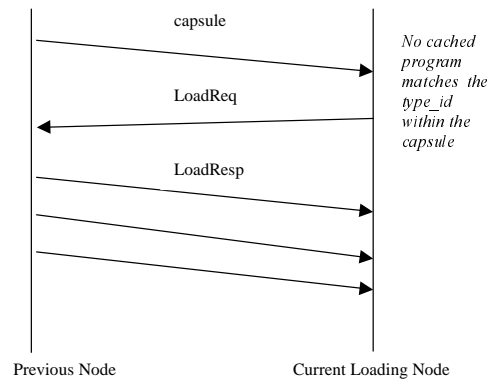


Figure 1 On-demand Code Load Procedure

2.3 An Active Network Node Architecture Based on ANTS

In order to realize the goal of collecting network information, we use ANTS to make the network active. The main reason is that the ANTS architecture is compatible with the current existing network infrastructure: ANTS provides an application level toolkit for building and dynamically deploying new network protocols and can execute as an application in the network node. The use of the general purpose Java language in ANTS is another reason to use ANTS as our active network platform: this makes it relatively easy to incorporate active networks into the current IP networks. In principle we use ANTS as our active network architecture, however, to better support our tasks, we have made some changes to this architecture.

The first change is applied to the code distribution scheme. From above we see that there are some advantages related to the on-demand code loading procedure, e.g. it limits the distribution of code to where it is needed, however the startup performance of ANTS capsules is relative poor, there exists a relative long delay for the "first" capsule to be processed. Although the performance of the sequences of capsules that follow the same path and require the same processing could be greatly improved, this is not well suited for the first capsule with a short program (all the code can be carried in one capsule). In order to enhance the startup performance of capsules with short programs, we add an associated code distribution method to the ANTS architecture. According to this method, capsules may carry the method to be used to process the capsule in an active node. This method omits the code loading protocol procedure used for the code propagation, and is particularly suitable for capsules with short programs.

As a simple approach, we define different values for the TYPE ID in the ANEP header[2] to distinguish the two

different code distribution methods, representing the two “different” EEs respectively. User applications may select the code distribution method according to the size of their applications’ code. For the associated code distribution method, each capsule still carries an identifier in the payload field, identifying the method used to process the capsule. This identifier is also used as an index for caching this code in the active node and processing the capsule. In order to decrease the time used for code loading and caching, when a capsule arrives at a node, the node will still check if the method used for the processing of this capsule is already available in this node according to the type identifier. If yes, the corresponding method will be invoked and the data in the capsule will be processed, otherwise the code carried in the capsule will first be loaded and cached and then the capsule will be processed using this just cached method. The maximum number of cached methods in one node can be defined according to the resources at this node, such as memory etc.

The second change is in the format of each capsule. Besides defining new values of TYPE ID in the capsule header, we also provide for the specification of certain network nodes by allowing intermediate node option fields inside the capsules. These intermediate node options have two meanings: in the capsules sent from the source (represented by InterNodeOptGoal), they mean the node address, at which the user wants to gather information; in the capsules back to the sender (represented by InterNodeOptResult), they mean where they have collected information, so when the user receives the capsule again, he can check from this part in the capsule if he has got the information at all the nodes he wanted. In order to process the capsules in each network node as fast as possible, these node address information are defined as option fields[2] with the structure of TLV (Type/ Length/ Value) in the header of each capsule. So at each node, just from the header of a capsule decisions can be made whether the capsule need to be processed further or not at this node. It is notable that putting intermediate node address information in the header of a capsule can decrease the processing time only for the associated code distribution method. This may cause that there is no program caching at some network nodes at all (e.g. at nodes where no information collection has been requested), which may lead to a failure for the code loading procedure using the on-demand load method (which always requests code from the previous node) [18]. However, in order to keep both code distributing methods consistent, for the on-demand code loading method we put intermediate node address information also in the header of a capsule as option fields defined in ANEP.

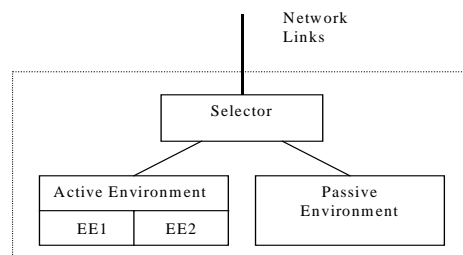


Figure 2 Network Node Architecture

Our architecture of the network nodes is shown in figure 2. There are three main components in this architecture:

Selector: a part of the router that enables the packets to select an active or passive environment. If the port number and the protocol type of a packet match a mask, this packet will be forwarded to the active environment, otherwise, to the passive environment.

Active Environment: Program code carried in the capsule will be executed in the active environment.

Passive Environment: This part of the router contains all the functions of the original router.

When a packet enters a network node, it will be forwarded to the active environment or the passive environment by the selector. In the active environment, the capsule will be handled according to the TYPE ID field in the ANEP header: either it is checked whether it is necessary to request the method for the processing of this capsule from the previously node, or the code carried in the capsule is cached and then the cached method is used to process this capsule.

3. Resource Collection in Active Networks

Our method of collecting network information was inspired by the well-know traceroute program. The traceroute program enables a user at a host to get a list of all the routers with the elapsing time to reach them, but it can only retrieve the hostname and the delay along a path. Together with active networks, packets can interrogate the network nodes and get more information about these nodes. For this purpose, we designed capsules that carry a program with the ability to calculate or acquire the related information about the corresponding network nodes with the help of the NodeOS. These capsules can traverse through the network and back to the source host with the desired network information. Figure 3 shows the path of the capsules.

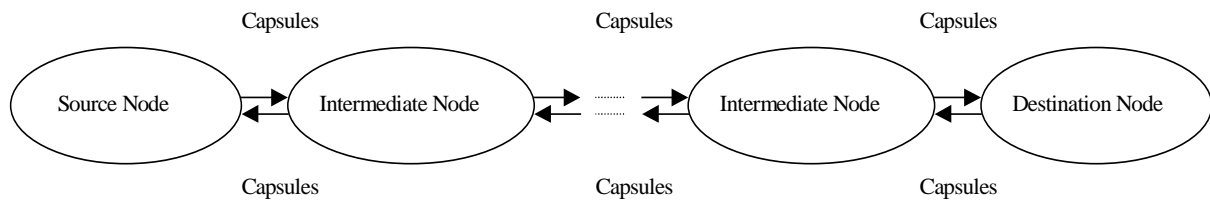


Figure 3 Capsule Path in the Network

In order to bring the desired network information to a user, the following information must be carried in the capsule from the user to the network and back to the user:

1. **Node address:** If a user has a special desire to acquire information only about some nodes, he may specify the addresses of these nodes in his capsule to the network.
2. **Algorithms:** In order to tell the active node what information and how to get the information, an according algorithms must be carried in the capsule from the user to the network.
3. **Result:** Certain information acquired from the nodes must be carried in the capsule back to the source user.

For the first point, we use the intermediate option fields (InterNodeOptGoal) in the capsule header to carry the node addresses from which a user wants to acquire the information, so that when an active node receives a capsule, it can first take a look if the capsule should be processed in this node and, hence, some information about the node specified in the capsule should be retrieved and added in the capsule to the next node. In the case that the user wants to get information about all nodes along the path to the destination, there are no intermediate node options of type InterNodeOptGoal.

The second point concerns user-specified requirements and these requirements are reflected by the program code carried in the capsule. The third point of the information is acquired from the node and hence is carried as normal data in the capsule. Of course the nodes' address information at which the capsule has retrieved the desired information will be added to the capsule header using intermediate node option fields (type InterNodeOptResult).

As a simple example, we suppose that we need the information about bandwidth and memory at all the active network nodes along the route from a source to the destination. For each of the bandwidth and memory information, we record three kinds of values: total amount, the amount used by this information collecting program and the amount still available. Besides these, we record also the transmission delay, node address etc, just like the normal "ping" and "traceroute" program. An example for the path taken by such collection capsules

and the content of the capsules at the various stages within the network is illustrated in figure 4. Following are some explanations of this figure.

1. We use 3 machines acting as active nodes in the example, they are located in Karlsruhe (129.13.42.152), Darmstadt (130.83.245.100) and Braunschweig (134.169.9.220) and represent source, intermediate network node and destination respectively. In each machine a routing table is stored. In order to compare the two code distribution methods, we suppose that when a capsule gets to the destination, it will go back to the source along the same nodes it used towards the destination. Hence on the way back to the source, the needed program is already cached in the network node. In this example, the information about node 130.83.245.100 is collected twice.
Note that the above three nodes are only active nodes in the way from the source to the destination. The actual nodes that the capsules pass in the real network are shown in figure 7 and 8 in Appendix.
2. The contents of the capsules shown in the figure are only accordance with the associated code distribution method. I.e. each of the capsules carries program code as it goes forward to the next node, even if it passes a node the second time, e.g. node 130.83.245.100. In the case that the on-demand code distribution method is used, in the first capsule to the next node (like node 130.83.245.100, 134.169.9.220) there is no program code, instead a code loading procedure as shown in figure 1 is used. On the way back to the source (e.g. in 130.83.245.100), there is no need for the code loading procedure, since the needed code has already been cached in this node.
3. The capsule from the last intermediate node back to the source does not carry program code (Here we suppose that the source maintains the code for the processing of the capsule till the end of the application). We consider here the last intermediate node as the node, at which the "next-hop" for a capsule according to the routing table is equal to the source address.
4. The meanings of some items in this figure are as follows:
Destination Option: destination option field as defined in ANEP[2], it has a structure of TLV (Type/

Length/ Value), and contains the destination of this capsule.

Source Option: source option field, defined by ANEP[2], contains the source of the capsule.

InterNodeOptResult (130.83.245.100): represents the intermediate option field in the header of a capsule, and has a similar structure as the destination/source option field defined in ANEP. It contains the corresponding network node address, here for example 130.83.245.100. We defined two types of the intermediate option field, in order to distinguish

the node at which a user wants to acquire information (represented by *InterNodeOptGoal*) and the node at which a user has got some information (represented by *InterNodeOptResult*).

Nbr: in order to record the network nodes sequences that a capsule has passed by, we add a number to each network node, including the destination node. Note that when a capsule passes a node the second time, though the node address is the same for both times, the number of the node is however different.

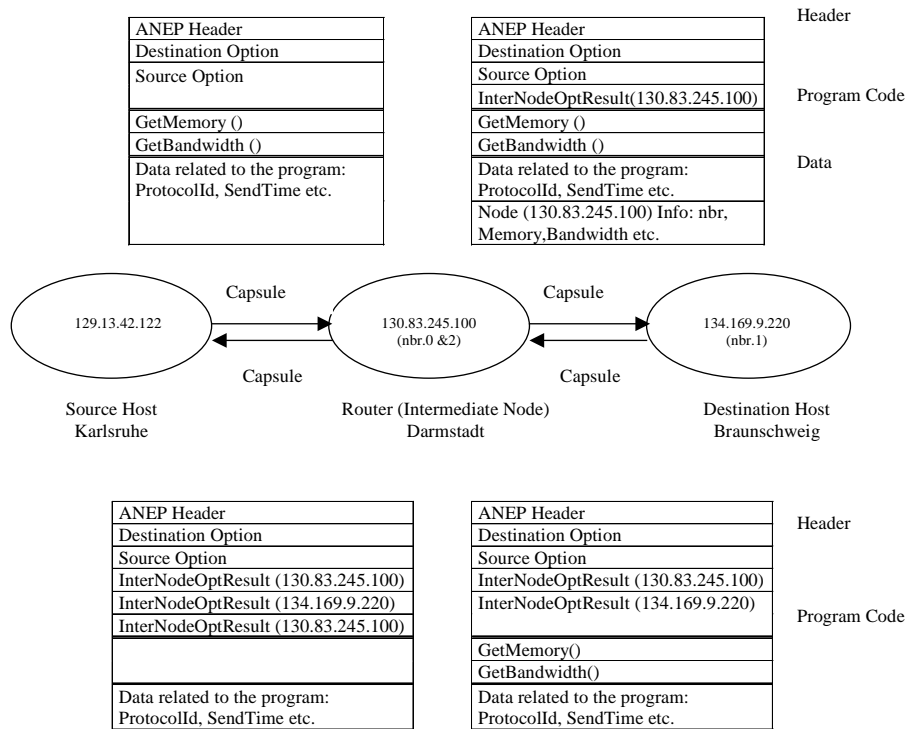


Figure 4 Capsule Path and Content in the Network

Figure 5 illustrates the user interface of our example. Using this interface, we can specify the number and interval of the capsules to be sent, as well as the addresses of nodes at which we want to get bandwidth and memory information. The results of the experiment can also be seen in this window.

Some remarks can be made using this example.

1. Bandwidth: static bandwidth can be seen as corresponding to the link capacity. The dynamic bandwidth can be calculated by sampling the counters that store accumulated bytes sent over the link. In this example we use static bandwidth to represent the total bandwidth of the nodes and use dynamic bandwidth to represent the bandwidth that is currently used by transferring this capsule. The difference of them represents the free bandwidth.
2. Memory: total memory means the memory provided by the Java environment.

3. For the moment, our methods for the calculation of memory and bandwidth are only simplified examples of the needed functionality.

In order to show the necessity and the performance of the associated code distribution method introduced above, we list some experimental results acquired from this example in figure 6 and examine the processing time of each capsule in the network node as well as the round trip delay of a capsule (represented by Total Transmission Delay in figure 6). The processing time is equal to the time interval between the arrival of a certain capsule and its departure time at a certain network node. It consists of time used for code loading, caching if necessary and the processing of a capsule using the cached code. The total transmission delay is the time interval between when a capsule leaves and when it returns to the source node. Therefore it consists of the processing time in the intermediate network nodes and the destination node as

well as of the transmission delay in different network sections. Because in practice there is a limit to the maximum length of each packet due to the physical link, e.g. for Ethernet 1500 bytes, the transfer of the code will be segmented into several packets if the length of the code exceeds the limitation. In order to study the relationship among code length, processing time, and code distribution

method, in this example we show the cases of different code length by stipulating of the code buffer length and UDP transmission buffer length in the program. In figure 6, N means the number of times in which the whole capsule will be transferred in UDP layer. Note that in this case even though N equals to 1 the code will also be transmitted in several packets in physical layer.



Figure 5 Simple User Interface for the Active Traceroute Application

Some results of our testing are listed in figure 6. Note that the values are the average values of the testing after some times (here we just want to compare the processing time in different situations, not evaluate the performance of the system). From figure 6 we see that the processing time of the first capsule is longer than that of the following capsule when the on-demand load code distribution method is used, i.e. the startup performance of the method is not so good, but the performance of the following capsules is very good: both processing time and total transmission delay are small. Using the associated method the difference between the first capsule and the following capsules is not large, but the processing time and total transmission delay are not as good as using the on-demand load method (for other than the first capsule),

except when the number of the segments of the program is equal to 1, i.e. all the code can be transferred in one capsule. In this case, the processing time and transmission delay for both the first capsule and the following capsules are relatively small. This means also, as expected, that the associated code distribution method is suited only to short programs. Note that there is still a small difference between the first capsule and the following capsules in the processing time and total transmission delay even in the case of associated load. The main reason is when a capsule reaches in a node, the identifier of the code carried in the capsule is first checked. If the code is not yet available (in the case of the first capsule), the code in the payload of the capsule needs to be cached. Otherwise the code is discarded.

	Code Segment Number N (in UDP Layer)	Node Number	1st Capsule	2nd Capsule	3rd Capsule	>10th Capsule
			Processing Time (ms)	Processing Time (ms)	Processing Time (ms)	Processing Time (ms)
On Demand Load (capsule length=3085 bytes)	N=1	1	53	0.667	1	1
		2	203	2.333	2.667	2
		3	5	2.667	2	2
		Total Trans. delay (ms)	1554	123	124	112
	N=6	1	105	2	2.667	0.667
		2	240	2.667	3	2.667
3		12	3.333	3	2.667	
Total Trans. delay (ms)	2676	132	125	128		
Associated Load (capsule length=5340 bytes)	N=1	1	4.667	2.333	2	2.333
		2	4.333	3	3	2.667
		3	3	2.667	2.333	2
		Total Trans. delay (ms)	194	171	169	168
	N=2	1	60	51	52	50
		2	65	55	56	54
		3	50	49	49	48
		Total Trans. delay (ms)	365	313	316	312
	N=6	1	219	214	207	212
		2	224	219	217	216
		3	211	213	209	213
		Total Trans. delay (ms)	1225	1122	1118	1119

Figure 6 Experimental Results

4. Related Work

Because of the importance of network information to applications, systems that focus on measurement of network status information have been developed by several groups. NWS[20] and Prophet[16] provide applications with benchmark-based predictions by sending messages to make communication measurements between pairs of computation nodes. These methods concentrated on the predictive accuracy and forecast of the performance of available resources but pay little attention to the requirements of the users. Globus[6] and Legion[7] provide support for a wide range of functions such as resource location and reservation, authentication and remote process creation mechanisms, but they do not focus on supporting application level access to network status information. Remos[4][8][9] is a system specially designed to collect different kind of information in networks. It uses different methods such as SNMP and benchmarks to collect network information and provides a standard interface format that is independent of the details of any particular type of network. One of the most important advantages of the system is that it provides a query-based interface and allows its applications to specify what kind of information they need. Remos can then use this application-specific information as a context to decrease their work only to provide these required information. Our approach can also provide this function: users can specify whatever information on which nodes they want. The Remos system provides the network information to the user in the format of a logical topology. Though it is more attractive and direct than other methods such as listing bandwidth or latency information for each pair of endpoints, however, to some extends, it cannot meet the needs of all users because different applications have different requirements. To this extent, our approach is more flexible: the user can specify the desired format of the network information in the program code sent to the network. This results from the active networks approach, which allows users to inject their program into the networks.

In recent years active networks has gained a rapid development. Much work related to enabling techniques, platforms, languages as well as reliability about active networks has been done in order to implement and integrate active networks into the current networks. In the same time, work about active applications also got a lot of attention. The most important application of active networks stems directly from their ability to program the network: new protocols and innovative cost-effective technologies can be easily deployed at intermediate network nodes[11]. There are also some specific applications that are beneficial from active networks, examples are congestion control, network management, multicasting and caching. Our work is to use active network techniques to collect information about networks

nodes, and then to detect the resource capabilities in these nodes, aiming to do resource management for active networks. The Darwin[14] system shares the same goal of using active network techniques, it introduces the concept of a delegate, a code segment that applications or service providers inject into the network to assist in the management of the network resources that are allocated to them. Darwin's delegates and our method have the same idea of "application-specific", i.e. in both cases the applications' method to solve network problems (resource management for Darwin or information collection for us) can be sent to the network in the form of a program and be adopted and executed within the network. The difference lies in that Darwin uses a programming interface to realize the application-specific program but we reach our goal through establishing a code execution environment. A similar method, namely through establishing an environment for the execution of programs, has also been used by the researchers in Bell Laboratories, but they use the Active Engine[12] to execute programs for the purpose of managing the network in a distributed way.

5. Conclusion

In this paper, we suggest a new method for the collection of network information. It is inspired by the well-known traceroute program and uses the gradually ripe active network techniques. By injecting and executing user programs in the network nodes, the collection of the network information becomes more freely and in a user-specific way. Through this method, users can specify both the content and the format of the network information that they want to acquire. Besides these, they can also specify nodes they want to gather information about.

Our implementation of this method is based on ANTS. However, by adding a new associated code distribution way, our active network environment provides also better performance for applications with short programs. The additional intermediate node option field in the header of each capsule makes our goal of collecting network information more flexible. Further work will be directed towards the practical usage, e.g. what kind of network information can be presented to the users etc. Besides this, we will study how this information can be used for the resource management of both passive and active networks.

Acknowledgements

We would like to thank Dr. Carsten Griwodz at Darmstadt University of Technology and Jidong Wu at Braunschweig University of Technology for their help with the setup of the test environment.

References

- [1] D.Alexander, W.A.Arbaugh, M.W.Hicks, P.Kakkar, A.D.Keromytis, J.T.Moore, C.A.Gunter, S.M.Nettles and J.M.Smith, "The SwitchWare Active Network Architecture", IEEE Network, May/June 1998.
- [2] D.Alexander, B.Braden, C.A.Gunter, A.W.Jackson, A.D.Keromytis, G.J.Minden and D.Wetherall, "Active Network Encapsulation Protocol (ANEP)", July 1997.
- [3] J.Case, K.McCloghrie, M.Rose and S.Waldbusser. "Structure of management information for version 2 of the simple network management protocol (SNMPv2)". RFC1902, January 1996.
- [4] Tony Dewitt, Thomas Gross, Bruce Lowekamp, Nancy Miller, Peter Steenkiste and Jaspal Subhlok, "ReMos:A Resource Monitoring System for Network Aware Applications", Tech. Rep. CMU-CS-97-194, Carnegie Mellon University, December,1997.
- [5] A.Fernando, B.Kummerfeld, A.Fekete and M.Hitchens, "A New Dynamic Architecture for an Active Network", IEEE OPENARCH 2000.
- [6] I.Foster and C.Kesselman, "Globus: A Metacomputing Infrastructure Toolkit", International Journal on Supercomputing Applications, vol.11, no.2, 1997.
- [7] A.Grimshaw, W.Wulf and Legion Team, "The Legion vision of a worldwide virtual computer. Communications of the ACM 40, 1(January 1997).
- [8] B.Lowekamp, N.Miller, D.Sutherland, T.Gross, P.Steenkiste and J.Subhlok "A Resource Query Interface for Network-Aware Applications", proceeding of IEEE symposium on High-Performance Distributed Computing, July, 1998.
- [9] Nancy Miller and Peter Steenkiste, "Collecting Network Status Information for Network-Aware Applications", IINFOCOM 2000.
- [10] E.L.Nygren, S.J.Garland and M.F.Kaashoek, "PAN: A High-Performance Active Network Node Supporting Multiple Mobile Code Systems", In proceedings of OpenArch '99. Pages 78-89. New York. March 27, 1999.
- [11] Konstantinos Psounis, "Active Networks: Applications, Security, Safety, and Architectures", IEEE Communications Surveys, First Quarter 1999.
- [12] D.Raz and Y.Shavitt, "Active Networks for Efficient Distributed Network Management", IEEE Communications Magazine, March 2000.
- [13] B.Scharte, "Smart Packets for Active Networks", Second International Conference on Open Architectures and Network Programming (OPENARCH), New York, 1999.
- [14] E.Takahashi, P.Steenkiste, J.Gao and A.Fisher, "A Programming Interface for Network Resource Management", Second International Conference on Open Architectures and Network Programming (OPENARCH), New York, 1999.
- [15] D.Tennenhouse and D.Wetherall, "Towards an Active Network Architecture", Proc. Multimedia Computing and Networking, San Jose, CA, 1996.
- [16] J.Weissman and X.Zhao, "Scheduling Parallel Applications in Distributed Networks", Cluster Computing, vol.1, no.1, pp.95-108, May 1998.
- [17] D.Wetherall, "Developing Network Protocols with the ANTS Toolkit", Design review, August 1997, <http://www.sds.lcs.mit.edu>.
- [18] D.Wetherall, J.Gutttag and D.Tennenhouse, "ANTS: A Toolkit for Building and Dynamically Deploying Network Protocols", IEEE OPENARCH'98, San Francisco, April 1998.
- [19] D.Wetherall, J.Gutttag and D.Tennenhouse, "ANTS: Network Services Without the Red Tape", IEEE Computer, 1999.
- [20] R.Wolski, N.Spring and C.Peterson, "Implementing a performance forecasting system for metacomputing: The network weather service", Tech.Rep.TR-CS97-540, University of San Diego, May 1997.
- [21] Y.Yemini and Sushil da Silva, "Towards Programmable Networks", proc. IFIP/IEEE Int'I Workshop on Distributed Systems, Operations, and Management, L'Aquila, Italy, 1996.

Appendix Capsules' route from Karlsruhe to Darmstadt, and to Braunschweig

We use the normal traceroute program provided by Unix system to monitor the nodes that the capsules pass in the real network. Because our example introduced in section 3 requires that all the capsules should pass the intermediate active node in Darmstadt (130.83.245.100), we executed traceroute program in Karlsruhe (source) and Darmstadt (intermediate node) respectively. Figure 7 shows the result of the traceroute from Karlsruhe to Darmstadt (130.83.245.100), and figure 8 is the result from Darmstadt to Braunschweig (134.169.9.220).

```

traceroute to 130.83.245.100 (130.83.245.100), 30 hops max, 40 byte packets
 1 tm-r70.telematik.informatik.uni-karlsruhe.de (141.3.70.254) 0.441 ms 4.331 ms 0.222
ms
 2 192.168.1.254 (192.168.1.254) 0.466 ms 0.437 ms 0.424 ms
 3 172.21.3.9 (172.21.3.9) 0.806 ms 0.747 ms 0.807 ms
 4 r-196er-rnz-164-22a.rz.uni-karlsruhe.de (129.13.196.248) 1.554 ms 0.940 ms 1.177
ms
 5 Karlsruhe1.BelWue.DE (129.143.167.5) 2.133 ms 1.455 ms 2.342 ms
 6 karlsruhe.core.xlink.net (129.143.167.106) 3.152 ms 3.053 ms 2.956 ms
 7 r100-pl.ka.de.KPNQwest.net (194.122.243.17) 5.423 ms 3.843 ms 3.005 ms
 8 r2-pl.ka.de.KPNQwest.net (194.122.243.9) 2.841 ms 2.435 ms 2.638 ms
 9 r1-erp1.f.de.KPNQwest.net (194.122.227.150) 8.561 ms 8.047 ms 8.588 ms
10 r1-decix.f.de.KPNQwest.net (194.122.242.138) 9.179 ms 8.728 ms 9.060 ms
11 r2-ixa1.f.de.KPNQwest.net (194.122.243.117) 9.843 ms 8.519 ms 9.517 ms
12 gw.tu-darmstadt.de (194.122.246.6) 9.972 ms 9.725 ms 9.598 ms
13 193.23.248.66 (193.23.248.66) 52.935 ms 13.052 ms 12.599 ms
14 TUD-router1.hrz.tu-darmstadt.de (193.23.248.1) 12.760 ms 14.759 ms 16.276 ms
15 rsm1252a-ip.hrz.tu-darmstadt.de (130.83.128.19) 14.272 ms 13.898 ms 19.237 ms
16 test04.kom.e-technik.tu-darmstadt.de (130.83.245.100) 17.817 ms * 27.062 ms

```

Figure 7 Actual network nodes from Karlsruhe (129.13.42.122) to Darmstadt (130.83.245.100)

```

traceroute to 134.169.9.220 (134.169.9.220), 30 hops max, 40 byte packets
 1 bagpipe (130.83.139.254) 0.425 ms 1.431 ms 0.365 ms
 2 cis1146-245-96.hrz.tu-darmstadt.de (130.83.245.126) 1.331 ms 1.296ms 1.199
ms
 3 cis12.hrz.tu-darmstadt.de (130.83.128.3) 2.764 ms 1.410 ms 1.156 ms
 4 manda-router2.hrz.tu-darmstadt.de (193.23.248.8) 3.246 ms 3.604 ms 3.706ms
 5 rt-f-tb1.hrz.tu-darmstadt.de (193.23.248.67) 3.740 ms 3.593 ms 3.401 ms
 6 r2-ixa1.f.de.KPNQwest.net (194.122.246.5) 8.420 ms 4.306 ms 4.050ms
 7 r1-decix.f.de.KPNQwest.net (194.122.243.113) 3.838 ms 3.989 ms 3.336 ms
 8 * * *
 9 cr-frankfurt1.g-win.dfn.de (188.1.80.37) 5.174 ms 6.342 ms 4.597 ms
10 cr-hannover1.g-win.dfn.de (188.1.18.13) 10.423 ms 11.956 ms 12.954ms
11 ar-braunschweig2.g-win.dfn.de (188.1.88.46) 16.472 ms 10.749 ms 11.410 ms
12 134.169.2.1 (134.169.2.1) 11.233 ms 11.433 ms 11.049 ms
13 rzpool05.rz.tu-bs.de (134.169.9.220) 13.605 ms 12.618 ms 13.596 ms

```

Figure 8 Actual network nodes from Darmstadt (130.83.245.100) to Braunschweig (134.169.9.220)