Towards Distributed Context Management in Ambient Networks

Christoph Reichert, Michael Kleis
Fraunhofer FOKUS
Kaiserin-Augusta-Allee 31
10589 Berlin, Germany
E-mail:{reichert,kleis}@fokus.fraunhofer.de

Raffaele Giaffreda
BT, Polaris House, Rm 129
Adastral Park, Martlesham Heath
Ipswich IP5 3RE, United Kingdom
E-mail: raffaele.giaffreda@bt.com

Abstract— **Context information makes applications and networks aware of their situation. A scalable and distributed architecture consisting of context sources, context processors, and context sinks as basic components is proposed, together with an outline of a client-server protocol between these components. Two important functions of this architecture are context coordination and context management, which can be carried out in a fully distributed and self-organized way by means of peer-to-peer overlay networks and recursive cascading of context processors. The basic properties of the architecture and its potential evolution are discussed.**

I. Introduction

Context information makes applications and networks aware of their situation [1] and improves certain functions in network or applications, but is typically not *critical* in the sense that a functionality cannot be provided at all if the context information is not available. However, context information is expected to play a vital role in *supporting* autonomic decision making and is regarded as a necessity for self-organization.

The Ambient Networks [2] vision supports spontaneous merging of heterogeneous networks through the implementation of an architecture based on a modular Ambient Control Space (ACS). Such an autonomic behavior requires decision-making capabilities to be implemented in support of each of the functions the ACS is made of. The main role of the ”context provisioning” module is to collect, process and manage context information on behalf of each of the ACS functions. The reason behind this requirement is twofold. First, the availability of context information helps reduce the scope of the problem that any autonomic decision-making algorithm is faced with. Second, it has been proven useful to separate context information collection, processing and management from its actual use [3].

The main challenge for a context architecture arises from the fact that basically *any* information about a network, its services, applications, nodes, links and users, can be valuable context information and should therefore be accessible to basically any other component in the network. The basic problem is scalability regarding these potentially large amounts of data. The guiding principles are therefore twofold: First, requests for context information are answered only on demand, and proactive context information processing is avoided. (Note that this approach does not exclude caching.) Second, distributed peer-to-peer overlay techniques are exploited to spread the work load over several nodes. Within the Ambient Networks project, the size of the ”context information space” has been further reduced through agreeing with the clients (other functions of the ACS) what could be considered useful context information needed to improve the operations of each of the functions [4].

This work outlines a system architecture for context retrieval, processing, coordination and management. Sec. II describes the basic components of the architecture, in particular context processors, sketches a protocol for context related conversation between these components, and proposes P2P overlay networks for distributed context coordination. Sec. III describes both a centralized and distributed approach for context management, accomplishing the task of establishing multipipes for context processing. As an example scenario, autonomic provider selection using out architecture is described in Sec. IV. Sec. V discusses the more important aspects of the architecture in more detail, and Sec. VI concludes with a summary and future work.

II. The Architecture

A. Context Association

The elemental concept of the architecture is the context association. A context association is a directed relation from a context *source* to a context *sink*, i.e., the direction is that of the context information flow. The context source is the component providing/producing the context information, and the context sink is the component using/consuming it.

A context association has certain attributes, among which are Context Level Agreements (CLAs), Quality of Context (QoC) specifications, and the protocol actually used to retrieve the context information. The protocol is assumed to be a client-server protocol, where the context source acts as the server, and the context sink acts as the client. Modes of retrieval

[1]This paper describes work undertaken in the Ambient Networks Information Society Technologies (IST) 507134 project. The IST program is partially funded by the Commission of the European Union. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the Ambient Networks project or the Context Aware Networks work package.

(server push vs. client pull) are also attributes of a context association.

Context sink is any entity embedding a context client (making the entity context-aware). Otherwise, we do not make any assumptions about context sinks, like when or why they request, how they are affected by, or what decisions are made based on context information.

B. Context Sources, CIBs and URIs

We assume that a context source provides descriptive context information in form of a structured set of information elements, stored-in or simply linked-to via the Context Information Base (CIB), similar in nature to an SNMP Management Information Base (MIB) [5]. The content and structure of a CIB are determined by its *type*, which may be a SNMP MIB, an XML Document Type Definition (DTD), a file format etc. A particular CIB instance of a given type is identified by its Uniform Resource Identifier (URI) [6]. The information represented by a CIB may be dynamic and change over time, but only the context source updates its CIB. The basic operations to retrieve context information are that a context client fetches the content of the CIB by means of a client-server protocol, or subscribes for event notifications delivered when the CIB content changes. A context source maintaining one or more CIBs accessible via a context protocol is also called Context Provider Agent (CPA) [7].

C. Context Coordinator

All context sources register their CIBs at a conceptually centralized entity, the Context Coordinator (ConCoord). The ConCoord is the first point of contact for a context client: clients query the ConCoord in order to get the locations of CIBs, and the ConCoord responds with contact information of the CIBs. In other words, the ConCoord does not store the context information itself, but pointers to it. The functions of the ConCoord are summarized as:

- a *registry* where a context source registers the URIs of its CIBs with its contact information. Context sources are authenticated.
- a *resolver* which maps a context information request to one or more URIs.
- a function to *authenticate and authorize* client access to CIBs.

A registered CIB is uniquely identified by its URI, and completely described by the triple (URI, type, contact).

The registry of the ConCoord is itself a CIB, the meta-CIB of all other CIBs. The meta-CIB should also be accessible by the protocol primitives described below, since this enables clients to subscribe to events like "notify me whenever a new CIB of type X registers". This is important information for context clients to detect new sources of context information, or to learn that currently used context sources are no longer available.

D. Context Protocol Primitives

A context protocol provides basically the following primitives:

- REGISTER. A context source registers the URIs of its CIBs and its contact information at the ConCoord.
- RESOLVE. A context client requests context information in form of URIs from the ConCoord, which responds with the contact information of the corresponding CIBs.
- FETCH. A context client fetches context information from a CIB.
- SUBSCRIBE/NOTIFY. A context client subscribes to a CIB with an event specification. Thereafter, the client receives notifications whenever the CIB changes in a way specified by the client.

Note that the FETCH primitive can only be used for descriptive context, while SUBSCRIBE/NOTIFY is used for events and descriptive context. Note also that there is no primitive to update a CIB, since this is done exclusively by the context source which "owns" its CIBs.

The motivation for RESOLVE is that a client can locate its desired CIBs once and issue FETCH requests or change subscriptions as often as desired without generating more activity in the ConCoord than necessary. The alternative would be to send all FETCH or SUBSCRIBE requests to the ConCoord, which than locates the CIBs and forwards the requests to them.

E. Context Processors

The above context protocol primitives and the architectural elements introduced so far allow clients to request only "raw" context information provided by CIBs. These do not provide yet a way to filter, aggregate, and correlate context information. This is accomplished by *context processors* (or *synthesizers* [8], which consist of three parts:

- a context client to get context information from one or more *input* CIBs of type $T_1, T_2, \ldots, T_n$.
- a *processing function* $f : T_1 \times T_2 \times \ldots T_n \mapsto T$ which transforms input context information of types T_i into output information of type T.
- an *output* CIB of type T which represents the processed information and makes it accessible via the context protocol.

A context processor is therefore a context client "back to back" to a context source with a processing function in between. Context processing is then performed in a data-driven manner following the *pipes-and-filters* pattern by associating the output of a processor with the input of another.

Fig. 1 depicts a directed acyclic graph (DAG) whose nodes are context sources, context processors (CP), and context clients and whose edges are context associations. Nodes with zero in-degree are *initial* CIBs and the original source of "raw" context information. Nodes with zero out-degree are *final* context clients and the ultimate sink of context information. Nodes with non-zero in-degree and non-zero out-degree are context processors. The subgraph for a client, i.e., the subgraph

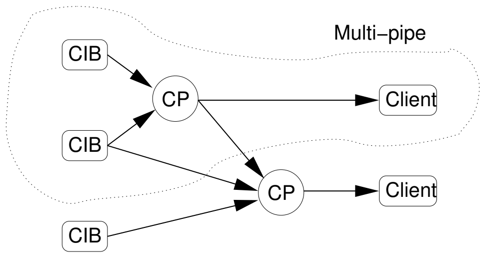

Fig. 1. A DAG constructed by initial context sources, context processors and final context sinks.

made up of all nodes and edges having a directed path to the client, is called the *multi-pipe* for that client.

Context processors with a single input are either *filters*, extracting specific information from their input, *converters*, transforming the input into another format, or *loggers*, recording the history of context information. Context processors with multiple inputs act as *aggregators* or *correlators* on their inputs.

Context processing can basically be performed in two modes. When a client issues a FETCH to get context information from a directly associated context processor, the FETCH request is propagated back towards the initial context sources. Alternatively, a context processor might be able to serve the request from a cache. Recursively fetching and processing context information is the *client-driven* mode (pull). In contrast, recursive subscription and notification is the *source-driven* mode (push).

Mixtures of both modes are also possible. For instance, a caching processor with a single input might exist solely for the purpose to move off load from another context source, which experiences a high rate of FETCH requests. The caching processor subscribes to the CIB and responds to FETCH requests on behalf of the initial context source.

Note that only the *type*, but not the *URI*, of the output CIB of a context processor is determined. The actual content of the output CIB and its URI depend on the CIB instances feeding the processor. For instance, when a processor receives input from user specific CIBs with user specific URIs, then the output is also user specific and should be identified by user-specific URIs.

F. Distributed Context Coordination

The registry of the ConCoord has to map context URIs to contact information of the CIBs identified by these URIs. The registry can be realized by Distributed Hash Tables (DHTs) using multiple schemes from the area of structured Peer-to-Peer (P2P) overlay networks like Chord [9], Content Addressable Networks (CAN) [10], or Tapestry [11]. In order to simplify the explanation, we assume that every context source, sink, and processor joins the P2P overlay, which makes up the distributed ConCoord.

After joining the P2P network, context sources register the URIs of their CIBs in form of pairs (URI, contact). The context source first applies a common, uniform hash function to the URI. The resulting hash key identifies the node on which the entry is to be stored. Then, the pair (URI, contact) is sent through the overlay to this node and stored there.

When any other overlay node issues a RESOLVE request for a given URI, the hash function is applied on the URI to identify the node where the entry is stored, and the entry is then retrieved via the overlay. (Messages internal to the P2P overlay are not described here and different from those in Sec. II-D.)

Methods to self-organize the nodes of a P2P network into a resilient overlay topology are well known and guarantee upper bounds for the scope of a search (e.g. $O(logN)$ in the case of Chord [9], where N is the number of nodes in the ConCoord), so that a distributed ConCoord is quite feasible.

III. Context Management

Context processors register their type with the ConCoord. The ConCoord's registry therefore maintains initial CIBs and context processors.

When the ConCoord receives a RESOLVE request, the ConCoord performs the following steps:

1) It looks up the registry whether there is an initial CIB for this request. If so, its contact information is returned.
2) Otherwise, the request is passed to a *context manager*, which tries to determine a multi-pipe whose final output CIB provides the desired context information.

The focus of this section is on approaches for step 2, context management, which means the task of instantiating multi-pipes in both centralized and distributed context management.

A. Centralized Context Management

A centralized context manager (CoMa) receives a RESOLVE request to instantiate a multi-pipe. The CoMa has access to a database which maps context types to *blueprints* of multi-pipes. A blueprint specifies

- which context processors are required, and
- how they have to be interconnected by context associations,

to provide the requested type of context information. In other words, a blueprint is a multi-pipe without initial input CIBs. A blueprint must be type consistent in the sense that a context processors output type must match the input type of the next one. Polymorphic processor functions might also be advantageous in some cases. In essence, a blueprint with multiple context processors is the functional composition of its constituent processing functions. As an example, for a linear pipe of processors with functions f, g, and h, the function of the blueprint is then $f \circ g \circ h = h(g(f(x)))$.

Note that as for a single processor, a blueprint specifies only the type of its input CIBs, but not their instances (URIs). The URIs for the initial input CIBs have to be inferred from the RESOLVE request.

There may be several different blueprints for basically the same type of context information, because there are often several different ways to obtain context information, with different levels of granularity, accuracy, etc. We expect that the right choice can often be made by taking Quality of Context (QoC) into account, so that both blueprints as well as RESOLVE requests have QoC specifications associated with them.

The basic steps during multi-pipe establishment are then as follows:

1) CoMa infers the context type from the RESOLVE request (URI and QoC) and looks up a blueprint for it. If this fails, a "context not available" error is returned.
2) Otherwise, establish appropriate context associations between the processors specified in the blueprint.
3) Derive the initial context sources and establish context associations between them and the first context processors to form the complete multi-pipe.

This central approach concentrates the knowledge about multi-pipes at a single entity, the CoMa, and requires this knowledge to be represented explicitely in form of blueprints as well as the mapping of context types to blueprints.

Given that the client knows best about the nature of the desired context, a further approach is to let the client specify the blueprint with required input CIBs. The client can check by accessing the meta-CIB whether the required processors are available. If so, it can include the blueprint into the RESOLVE request.

B. Distributed Context Management

A more interesting approach is to enable distributed context management in a self-organized way, circumventing the need for a centralized CoMa. Recall that a processor function is strongly typed. A type signature like $f : T_1 \times T_2 \mapsto T$ of a processor basically says: "I can provide context information of type T, given that my inputs are of type T_1 and T_2."

This enables to establish a multi-pipe recursively and in a distributed way. The ConCoord locates the final context processor, which locates the processors for its input, which again locate the processors for their input, and so on, until the inputs are all initial CIBs. More precisely, the following steps occur:

1) A client sends a RESOLVE request with a context URI to the ConCoord.
2) The ConCoord checks whether the URI identifies an initial CIB. If so, it returns the contact information of the CIB.
3) Otherwise, the ConCoord infers the type of the required CIB, locates a processor with this type as its output type, and passes the URI of the original RESOLVE request to this processor.
4) The processor infers the URIs of its input and sends a RESOLVE request with these URIs to the ConCoord.
5) Steps 2 – 4 are repeated until all URIs can be resolved during step 2.

Fig. 2 depicts the message sequence chart for the recursive establishment of a linear pipe between a client, two context processors, and an initial CIB, as well as the first FETCH of the client.[1]

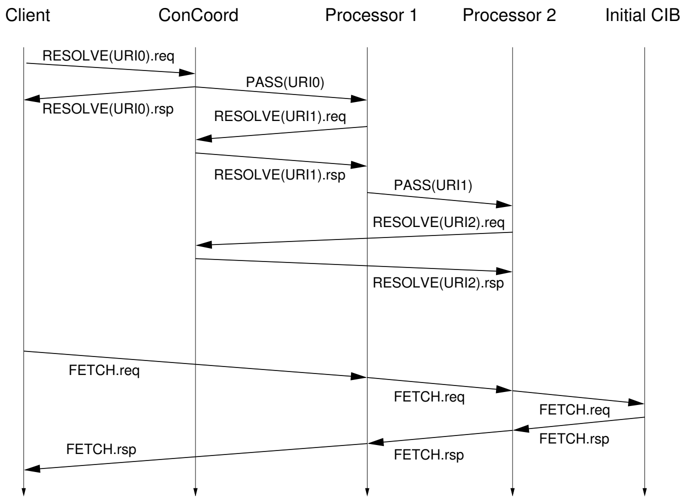

Fig. 2. Recursive establishment of a linear multi-pipe.

The critical requirement for the recursive approach is that a processor, knowing already its input types, is also able to infer which CIB instances of that type, i.e., their URIs, are required. This basically says: "I can provide context identified by URI X (of type T), given that my input CIBs are URI Y_1 (of type T_1) and Y_2 (of type T_2)." This requirement is much stronger than just inferring the input types.

C. Inter-domain Context Management

A context client in one domain should be able to ask for context information about another domain, which requires to consult the ConCoord of the other domain. The remote ConCoord has to be located (or a representative node in the distributed overlay), and we propose to define a new DNS SRV record [12] for the resolution of ConCoords.

A client always asks its local ConCoord even for remote context URIs. It is then the task of the local ConCoord to locate and contact the remote ConCoord and relaying the RESOLVE request to it, maybe after a Context Level Agreements (CLA) has been established between both ConCoords. This design decision is motivated by the goal to keep context clients as simple as possible, and also enables caching of both remote ConCoord locations and CLAs by the local ConCoord.

IV. Example Scenario

One possible application of our architecture is the following: Consider a mobile network (train, airplane, car) reaching an area where two or more providers offer WLAN access (train station, airport, city). Each provider offers access with different properties (bandwidth, QoS, price, etc.), and the mobile network contains an agent which selects the provider automatically (maybe based on policies). The agent needs two

[1]PASS is a protocol message internal to the context architecture to pass the received URI to the next processor. It is never received or sent by final clients or initial sources.

types of network context information: first, it needs to be notified whenever an alternative provider becomes available, and second, that provider's access properties.

Our context architecture supports this scenario as follows: The agent first subscribes to the set of all reachable providers to be notified whenever the new provider is available. The set of available providers is maintained by a context processor, which itself subscribed to all wireless interfaces of the mobile network (excluding devices of its users), which detected the access point of the new provider. When the agent is notified, it asks its local ConCoord to resolve the URI for access properties. The local ConCoord forwards this request to the remote ConCoord of the new provider. After the resolution is complete, the agent fetches the access information and finally makes the decision.

This example also shows that network context as understood in this work is typically *not* fuzzy or uncertain, which is often the case for user context [8].

V. Discussion

We argue that the described architecture contains many alternative designs as special cases, especially all centralized designs. In some networks, it is feasible to run a centralized ConCoord, centralized CoMa, and *all* context processors on a single node, which could be called a "context server". For small networks, this is certainly a reasonable approach, which also enables to reduce communication overhead by replacing the protocols by local software interfaces.

We do *not* propose a fine-grained design of context processors, albeit the architecture does not exclude such an approach, because it is likely to result in multi-pipes with large diameters (meaning here the longest path from an initial source to a final sink). The motivation for multi-pipes composed from context processors is to reduce communication and computation overhead by re-using intermediate results during context processing. Assume for the moment there were no context processors, and a client requires the information of several "raw" CIBs to make a decision, then context processing happens *within* the client, whether this is called so or not. When another client requires the same inputs for a similar decision, it repeats what the first client already did. The obvious approach is to "factor out" this commonality, which leads to the concept of context processors. Indeed, whenever such commonalities are recognized, it is the right time to ask: "Should this commonality become a dedicated context processor?" The answer to this question depends on how *frequent* such a processor will be used, and how *complex* its processing function is. High usage frequency and complexity are definitely a strong indication for a dedicated processor.

Context URIs play a key role in the proposed architecture. We basically assume that when a client wants specific context information, it is able to construct a URI for a CIB from which the information can be retrieved.[2] This assumption is justified by the fact that context information will be structured according to ontologies described in languages like the Web Ontology Language (OWL), which require that every concept in an ontology can be referenced by a unique URI [13]. For the recursive establishment of multi-pipes, an additional issue is that a context processor is able to infer the URIs of its input CIBs from its functional signature and the requested URI. Otherwise, context processors are also required to interpret ontologies.

[2]It is thereby irrelevant whether the CIB is an initial CIB or the end of a multi-pipe.

VI. Conclusion and Future Work

We presented an architecture for context processing based on context sources, context sinks, and context processors, and sketched the basic primitives required by a context protocol. In order to achieve distributed and self-organized context coordination and management, we proposed Distributed Hash Tables based on peer-to-peer overlay networks and recursive establishment of context processing multi-pipes. We argued that the proposed architecture enables implementations within the entire spectrum from a single, centralized context server for small ambient networks up to a fully distributed and self-organized system. Regarding the question what functionality should be provided by context processors, an evolutionary approach is proposed based on the usage frequency and complexity of the processor function. We then presented a real world scenario showing how network context can automate the selection of providers. Finally, we addressed the role of the context URI namespace.

References

[1] A. K. Dey, "Understanding and Using Context," *Personal and Ubiquitous Computing*, vol. 5, no. 1, pp. 4–7, Feb. 2001.

[2] N. Niebert, *et al.*, "Ambient networks: An architecture for communication networks beyond 3G," *IEEE Wireless Commun. Mag.*, vol. 11, no. 2, pp. 14–22, Apr. 2004.

[3] A. K. Dey, *et al.*, "A Conceptual Framework and a Toolkit for Supporting the Rapid Prototyping of Context-Aware Applications," *Human Computer Interaction*, vol. 16, no. 2–4, 2001.

[4] A. Jonsson *et al.*, "Ambient Networks ContextWare," Public Deliverable D6.1 of the EU Ambient Networks project, to be published on www.ambient-networks.org.

[5] D. Harrington, *et al.*, "An Architecture for Describing Simple Network Management Protocol (SNMP) Management Frameworks," RFC 3411, IETF, Dec. 2002.

[6] T. Berners-Lee, *et al.*, "Uniform Resource Identifier (URI): Generic Syntax," RFC 3986, IETF, Jan. 2005.

[7] M. Khedr *et al.*, "Negotiating Context Information in Context-Aware Systems," *IEEE Intell. Syst.*, vol. 19, no. 6, pp. 21–29, Nov./Dec. 2004.

[8] A. Ranganathan, *et al.*, "Reasoning about Uncertain Contexts in Pervasive Computing Environments," *IEEE Pervasive Computing*, vol. 3, no. 2, pp. 62–70, June 2004.

[9] I. Stoica, *et al.*, "Chord: A Scalable Peer-To-Peer Lookup Service for Internet Applications," in *ACM SIGCOMM*, San Diego, CA, USA, Aug. 2001, pp. 149–160.

[10] S. Ratnasamy, *et al.*, "A Scalable Content-Addressable Network," in *ACM SIGCOMM*, San Diego, CA, USA, Aug. 2001, pp. 161–172.

[11] B. Y. Zhao, *et al.*, "Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Routing," UC Berkeley, Tech. Rep. UCB/CSD-01-1141, Apr. 2001.

[12] A. Gulbrandsen, *et al.*, "A DNS RR for specifying the location of services (DNS SRV)," RFC 2782, IETF, Feb. 2000.

[13] J. Heflin, "OWL Web Ontology Language Use Cases and Requirements," W3C Recommendation, Feb. 2004. [Online]. Available: www.w3.org/TR/webont-req/