An Ambient ASM Model of Client-to-Client Interaction via Cloud Computing and an Anonymously Accessible Docking Service

Károly Bósa(✉)

Christian Doppler Laboratory for Client-Centric Cloud Computing, JKU, Linz, Softwarepark 21, 4232 Hagenberg im Mühlkreis, Austria
k.bosa@cdcc.faw.jku.at

Abstract. In our former work we have given a high-level formal model of a cloud service architecture in terms of a novel formal method approach which combines the advantages of the mathematically well-founded software engineering method called *abstract state machines* and of the calculus of mobile agents called *ambient calculus*. This paper presents an extension for this cloud model which enables client-to-client interaction in an almost direct way, so that the involvement of cloud services is transparent to the users. The discussed solution for transparent use of services is a kind of switching service, where registered cloud users communicate with each other, and the only role the cloud plays is to switch resources from one client to another. We also show in an example at the end of this paper how our novel client-to-client interaction mechanism can be utilized for the development of the anonymously accessible cloud services.

Keywords: Cloud computing · Ambient abstract state machines · Ambient calculus

1 Introduction

In [1] we proposed a new formal method approach which is able to incorporate the major advantages of the *abstract state machines (ASMs)* [2] and of *ambient calculus* [3]. Namely, one can describe formal models of distributed systems including mobile components in two abstraction layers such that while the algorithms of executable components (*agents*) are specified in terms of ASMs; their communication topology, locality and mobility described with the terms of ambient calculus in our method.

In [4] we presented a high-level formal model of a cloud service architecture in terms of this new method. In this paper, we extended this formal model with a *Client-to-Client Interaction (CTCI)* mechanism via a cloud architecture. Our envisioned cloud feature can be regarded as a special kind of services we call *channels*, via which registered cloud users can interact with each other in almost

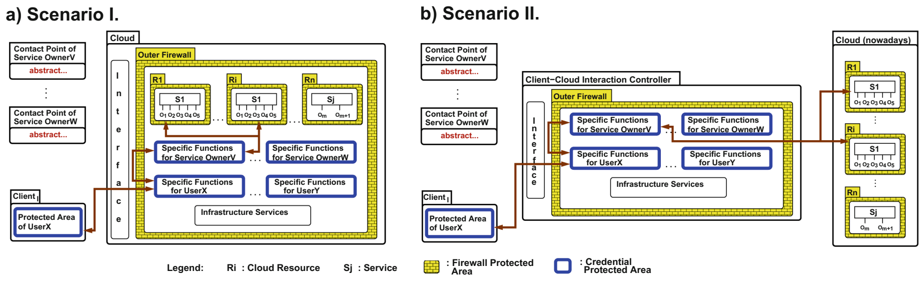
a) Scenario I.
b) Scenario II.
Contact Point of Service OwnerV
abstract...
Contact Point of Service OwnerW
abstract...
Client
Protected Area of UserX
Cloud
Interface
Outer Firewall
R1
Ri
Rn
S1
Sj
Specific Functions for Service OwnerV
Specific Functions for Service OwnerW
Specific Functions for UserX
Specific Functions for UserY
Infrastructure Services
Client–Cloud Interaction Controller
Cloud (nowadays)
Legend: Ri : Cloud Resource Sj : Service
: Firewall Protected Area
: Credential Protected Area

Fig. 1. Application of our model according to different scenarios.

direct way and, what is more, they are able to share available cloud resources among each other as well.

Some use cases, which may claim the need of such CTCI functions, can be for instance: dissemination of large or frequently updated data whose direct transmitting meets some limitations; or connecting devices of the same user (in the later case an additional challenge can be during a particular interpretation of the modeled CTCI functions, how to wrap and transport local area protocols, like *upnp* via the cloud).

The rest of the paper is organized as follows. Section 2 informally summarizes our formerly presented high-level cloud model. Section 3 gives a short overview on the related work as well as ambient calculus and ambient ASM. Section 4 introduces the definitions of some non-basic ambient capability actions which are applied in the latter sections. Section 5 describes the original model extended with the CTCI functions. Section 6 demonstrates how the CTCI architecture and the shared cloud resources can be applied for anonym usage of certain cloud services. Finally, Sect. 7 concludes this paper.

2 Overview on Our Model

Roughly our formal cloud model can be regarded as a pool of resources equipped with some infrastructure services, see Fig. 1a. Depending whether these abstract resources represent only physical hardware and virtual resources or entire computing platforms the model can be an abstraction of *Infrastructure as a Service (IaaS)* or *Platform as a Service (PaaS)*, respectively. The basic hardware (and software) infrastructure is owned by the cloud provider, whereas the softwares running on the resources are owned by some users. We assume that these softwares may be offered as a *service* and thus used by other users. Accordingly, we apply a relaxed definition of the term service cloud here, where a user who owns some applications running on some cloud resources may become a software service provider at the same time. Thus, from this aspect the model can be regarded as an abstraction of a mixture of *Software as a Service (SaaS)* and of IaaS (or a mixture of SaaS and PaaS).

We make a distinction between two kinds of cloud users. The normal *users* are registered in the cloud and they subscribe to and use some (software) services available in the cloud. The *service owners* are users as well, but they also rent some cloud *resources* to deploy some *service instances* on them.

For representing service instances, we adopt the formal model of *Abstract State Services (AS^2s)* [5,6]. In an AS^2 we have views on some hidden database layer that are equipped with *service operations* denoted by unique identifers $o_1,\ldots,o_n$. These service operations are actually what are exported from a service to be used by other systems or directly by users. The definition of AS^{2}s also includes the *pure data services* (service operations are just database queries) and the *pure functional services* (operation without underlying database layer) as extreme cases.

In our approach the model assumes that each service owner has a dedicated contact point which resides out of the cloud. It is a special kind of client that can also act as a server for the cloud itself in some cases. Namely, if a registered cloud user intends to subscribe to a particular service, she sends a subscription request to the cloud, which may forward it to such a special kind of client belonging to the corresponding service owner. This client responses with a special kind of action scheme called *service plot*, which algebraically defines and may constrain how the service can be used by the user[1]. (E.g.: it determines the permitted combination of service operations). This special kind of client is abstract in the current model.

The received service plots, which may be composed individually for each subscribing user by service owners, are collected with other cloud functions available for this particular user in a kind of personal user area by the cloud. Later, when the subscribed user sends a service request, it is checked whether the requested service operations are allowed by any service plot. If a requested operation is permitted then it is triggered to perform, otherwise it is blocked as long as a plot may allow to trigger it in the future. Each triggered operation request is authorized to enter into the user area of the corresponding service owner to whom the requested service operation belongs. Here a scheduler mechanism assigns to the request a one-off access to a cloud resource on which an instance of the corresponding service runs. Then the service operation request is forwarded to this resource, where the request is processed. Finally, the outcome of the performed operation returns to the area of the initiator user, where the outcome is either stored or send further to a given client device. In this way, the service owners have direct influence to the service usage of particular users via the provided service plots.

Regarding our proposed cloud model one of the major questions can be whether it is adaptable to the leading cloud solutions (e.g.: Amazon S3, Microsoft Azure, IBM SmartCloud, etc.). Since due to the applied ambient concept the relocation of the system components is trivial, we can apply our model according to different scenarios. For instance, all our novel functions including the client-to-client interaction can be shifted to the client side and wrapped into a

[1] For an algebraic formalization of plots *Kleene algebras with tests (KATs)* [7] has been applied.

Table 1. Definition of ambient calculus.

A. The Mobility	and Communication Primitives
P, Q, R::=	processes
$P \mid Q$	parallel composition
$n[\,P\,]$	ambient
$(\nu\ n)P$	restriction of name n within P[a]
0	inactivity (**skip** process)
$!P$	replication of P
$M.P$	(capability) action M then P
$(x).P$	input action (the input value is bound to x in P)
$\langle a \rangle$	async output action
$M_1. \ \ldots \ .M_k$	a path formation on actions
M::=	capabilities
$\textsc{In}\ n$	entry capability (to enter n)
$\textsc{Out}\ n$	exit capability (to exit n)
$\textsc{Open}\ n$	open capability (to dissolve n's boundary)

B. Reduction (Operational Semantics)
$P \equiv P', Q \equiv Q', P \longrightarrow Q \Longrightarrow P' \longrightarrow Q'$
$P \longrightarrow Q \Longrightarrow P \mid R \longrightarrow Q \mid R$
$P \longrightarrow Q \Longrightarrow n[\,P\,] \longrightarrow n[\,Q\,]$
$P \longrightarrow Q \Longrightarrow (\nu\ n)P \longrightarrow (\nu\ n)Q$
$n[\,\textsc{In}\ m.P \mid Q\,] \mid m[\,R\,] \longrightarrow m[\,n[\,P \mid Q\,] \mid R\,]$
$m[\,n[\,\textsc{Out}\ m.p \mid Q\,] \mid R\,] \longrightarrow n[\,P \mid Q\,] \mid m[\,R\,]$
$\textsc{Open}\ n.P \mid n[\,Q\,] \longrightarrow P \mid Q$
$(x).P \mid \langle a \rangle \longrightarrow P(x/a)$

C. Structural Congruence (Operational Semantics)
$P \equiv P$
$P \equiv Q \Longrightarrow Q \equiv P$
$P \equiv Q, Q \equiv R \Longrightarrow P \equiv R$
$P \equiv Q \Longrightarrow \P \mid R \equiv Q \mid R$
$P \equiv Q \Longrightarrow n[\,P\,] \equiv n[\,Q\,]$
$P \equiv Q \Longrightarrow !P \equiv !Q$
$P \equiv Q \Longrightarrow (\nu\ n)P \equiv (\nu\ n)Q$
$P \equiv Q \Longrightarrow M.P \equiv M.Q$
$P \equiv Q \Longrightarrow (x).P \equiv (x).Q$
$P \mid Q \equiv Q \mid P$
$(P \mid Q) \mid R \equiv P \mid (Q \mid R)$
$!P \equiv P \mid !P$
$(\nu\ n)(\nu\ m)P \equiv (\nu\ m)(\nu\ n)P$
$(\nu\ n)(P \mid Q) \equiv P \mid (\nu\ n)Q \quad \texttt{if}\ n \notin fn(P)$
$(\nu\ n)(m[\,P\,]) \equiv m[\,(\nu\ n)P\,] \quad \texttt{if}\ n \neq m$
$P \mid 0 \equiv P$
$!0 \equiv 0$
$(\nu\ n)0 \equiv 0$

[a]Name Restriction creates a new (unique) name n within a scope P. One must be careful with the term $!(\nu\ n)P$, because it provides a fresh value for each replica, so $(\nu\ n)!P \neq !(\nu\ n)P$.

middleware software which takes place between the end users and cloud in order to control the interactions of them, see Fig. 1b. The specified communication topology among the distributed system components remains the same in this later case.

3 Related Work

It is beyond the scope of this paper to discuss the vast literature of formal modeling mobile systems and SOAs, but we refer to some surveys on these fields [8–10].

One of the first examples for representing various kinds of published services as a pool of resources, like in our model, was in [11].

In [12] a formal high-level specifications of service cloud is given. This work is similar to ours in some aspects. Namely, it applies the language-independent AS^{2}s with algebraic plots for representing services. But it principally focuses on service specification, service discovery, service composition and orchestration of service-based processes; and it does not apply any formal approach to describe either static or dynamically changing structures of distributed system components.

Another approach similar to ours is *Cloud Calculus* [13], which also uses ambient calculus for capturing the dynamic topology of cloud computing systems. Cloud Calculus is very effective to verify whether global security policies are preserved after virtual machine migrations, but it is a very specific tool which is not applicable for giving the formal specification of functionalities of cloud/distributed systems.

In the rest of this section, we give a short summary on ambient calculus and ambient ASM, respectively, in order to facilitate the understanding of the latter sections.

3.1 Ambient Calculus

The ambient calculus was inspired by the π-calculus [14], but it focuses primarily on the concept of locality and process mobility across well defined boundaries instead of channel mobility as π-calculus. The concept of *ambient* stands in the center of the calculus, see a summary of the definition of ambient calculus in Table 1.

The ambient calculus includes only the mobility and communication primitives depicted in Table 1A. The main syntactic categories are *processes* (including both ambients and agents) and *actions* (including both *capabilities* and *communication primitives*). A reduction relation $P \longrightarrow Q$ describes the evolution of a term P into a new term Q (and $P \rightarrow^* Q$ denotes a reflexive and transitive reduction relation from P to Q).

An ambient is defined as a bounded place where computation happens. An ambient is written as $n[P]$, where n is its name, which can be used to control access (entry, exit, communication, etc.), and a process P is running inside its *body* (P may be running even if n is moving). Ambient names may not be unique. Ambients can be embedded into each other such that they can form a hierarchical tree structure. An ambient body is interpreted as the parallel composition of its elements (its local ambients and its local agents) and can be written as follows:

$$n[\ P_1 \ | \ \dots \ | \ P_k \ | \ m_1[\dots] \ | \ \dots | \ m_l[\dots] \ ] \text{ where } P_i \neq m_i[\dots]$$

An ambient can be moved. When an ambient moves, everything inside it moves with it (the boundary around an ambient determines what should move together with it). An action defined in the calculus can precede a process P. P cannot start to execute until the preceding actions are performed. Those actions that are able to control the movements of ambients in the hierarchy or to dissolve ambient boundaries are restricted by capabilities. By using capabilities an ambient can allow some processes to perform certain operations without publishing its true name to them (see the entry, exit and open in Table 1). In case of the modeling of a real life system, communication of (ambient) names should be rather rare, since knowing the name of an ambient gives a lot of control over it. Instead, it should be common to exchange restricted capabilities to control interactions between ambients (from a capability the ambient name cannot be retrieved).

3.2 Ambient ASM

In [15] the ambient concept (notion of "nestable" environments where computation can happen) is introduced into the ASM method. In that article an ASM machine called MobileAgentsManager is described as well, which gives a natural formulation for the reduction of three basic capabilities (Entry, Exit and Open) of ambient calculus in terms of the *ambient ASM* rules. For this machine an ambient tree hierarchy is always specified initially in a dynamic derived function called *curAmbProc*. The machine MobileAgentsManager transforms the current value of *curAmbProc* according to the capability actions given in *curAmbProc*. Since one of the main goals of [15] is to reveal the inherent opportunities of the new ambient concept introduced into ASMs, the presented definitions for moving ambients are unfortunately incomplete.

Table 2. A Summary of the definitions of some non-basic capabilities.

Names	**New Reduction Relations (Based on the Definitions)**	**Definitions of the New Capabilities**
1) Renaming	$n[\ n\ \textsc{Be}\ m.P \mid Q\ ] \longrightarrow^{*} m[\ P \mid Q\ ]$	$n\ \textsc{Be}\ m.P \equiv (\nu\ s)(s[\ \textsc{Out}\ n \mid m[\ \textsc{Open}\ n.\textsc{Out}\ s.P\ ]\ ] \mid \textsc{In}\ s.\textsc{In}\ m)$
2) Seeing	$n[\,] \mid \textsc{See}\ n.P \longrightarrow^{*} n[\,] \mid P$	$\textsc{See}\ n.P \equiv (\nu\ r, s)(\ r[\ \textsc{In}\ n.\textsc{Out}\ n.r\ \textsc{Be}\ s.P\ ] \mid \textsc{Open}\ s\ )$
3) Wrapping	$n[\ m\ \textsc{Wrap}\ n.P\ ] \longrightarrow^{*} m[\ n[\ P\ ]\ ]$	$m\ \textsc{Wrap}\ n.P \equiv$ $(\nu\ s, r)(\ s[\ \textsc{Out}\ n.\textsc{See}\ n.s\ \textsc{Be}\ m.r[\ \textsc{In}\ n\ ]\ ] \mid \textsc{In}\ s.\textsc{Open}\ r.P\ )$
4) Allowing Code	$\textsc{Allow}\ key.P \mid key[\ Q\ ] \longrightarrow^{*} P \mid Q$	$\textsc{Allow}\ key.P \equiv \textsc{Open}\ key.P$
5) Drawing in (an Ambient)	$m[\ Q \mid \textsc{Allow}\ key\ ] \mid n[\ n\ \textsc{DrawIn}_{key}\ m.P\ ]$ $\longrightarrow^{*} n[\ Q \mid P\ ]$	$n\ \textsc{DrawIn}_{key}\ m.P \equiv$ $key[\ \textsc{Out}\ n.\textsc{In}\ m.\textsc{In}\ n\ ] \mid \textsc{Allow}\ m.P$
6) Drawing in Then Release a Lock	$m[\ Q \mid \textsc{Allow}\ key\ ] \mid$ $n[\ \textsc{DrawIn}_{key}\ m\ \textsc{ThenRelease}\ lock.P\ ]$ $\longrightarrow^{*} lock[\ n[\ Q \mid P\ ]\ ]$	$n\ \textsc{DrawIn}_{key}\ m\ \textsc{ThenRelease}\ lock.P \equiv$ $key[\ \textsc{Out}\ n.\textsc{In}\ m.\textsc{In}\ n\ ] \mid \textsc{See}\ m.lock\ \textsc{Wrap}\ n.\textsc{Allow}\ m.P$
7) Concurrent Server Process	$m[\ Q \mid \textsc{Allow}\ key\ ] \mid \textsc{Server}^{n}_{key}\ m.P$ $\longrightarrow^{*} \textsc{Server}^{n}_{key}\ m.P \mid n^{uniq}_{k}[\ Q \mid P\ ]$	$\textsc{Server}^{n}_{key}\ m.P \equiv$ $(\nu\ next)(next[\,] \mid$ $!(\nu\ n)(\textsc{Open}\ next.n[$ $n\ \textsc{DrawIn}_{key}\ m\ \textsc{ThenRelease}\ next.P\ ]))$

In [1] we extended this ASM machine given in [15], such that it fully captures the calculus of mobile agents and it can interpret the agents' algorithms (given in terms of ASM syntax in *curAmbProc* as well) in the corresponding contexts. By this one is able to describe formal models of distributed systems including mobile components in the mentioned two abstraction layers.

Since the definition of ambient ASM is based upon the semantics of ASM without any changes, each specification given this way can be translated into a traditional ASM specification.

Ambient ASM is not the only research which aims to build in a concept of mobile ambients to the ASM method. In [16] some advantages of a simple ambient concept introduced into ASM are demonstrated. Although this work was also inspired by ambient calculus, it is by far not refined and versatile as ambient ASM.

4 Definitions

As Cardelli and Gordon showed in [3] the ambient calculus with the three basic capabilities (Entry, Exit and Open) is powerful enough to be Turing-complete. But for facilitating the specification of such a compound formal model as a model of a cloud infrastructure, we defined some new *non-basic capabilities* encoded in terms of the three basic capabilities. Table 2 summarizes the definitions of these non-basic capabilities.

Below we give an informal description of each non-basic capability in Table 2. It is beyond the scope of the paper to present detailed explanations and reductions of their ambient calculus-based definitions, but we refer to our former works [4,17] for more details.

1. Renaming. This capability is applied to rename an ambient comprising this capability. Such a capability was already given in [3], but our definition differs from Cardelli's definition. In the original definition, the ambient m was not enclosed into another, name restricted ambient (it is called s in our definition), so after it has left ambient n, n may enter into another ambient called m (if more than one m exists as sibling of n).

2. Seeing. This operation was defined in [3] and it is used to detect the presence of a given ambient.

3. Wrapping. Its aim is to pack an ambient comprising this capability into another ambient.

4. Allowing Code. This capability is just a basic OPEN capability action. It is applied if an ambient allows/accepts an ambient construct (which may be a bunch of foreign codes) contained by the body of one of its sub-ambients (which may was sent from a foreign location). The name of the sub-ambient can be applied for identifying its content, since its name may be known only by some trusted parties.

5. Draw in (an Ambient). The aim of this capability is to draw in a particular ambient (identified by its name) into another ambient (which contains this capability) and then to dissolve this captured ambient in order to access to its content. For achieving this, a mechanism (contained by the ambient *key*) is applied which can be regarded as an abstraction of a kind of protocol identified by *key*. The ambient *key* enters into one of the available target ambients which should accept its content in order to be led into the initiator ambient.

6. Draw in then Release a Lock. This capability is very similar to the previous one, but after m has been captured by n (and before m is dissolved), n is wrapped by another ambient. The new outer ambient is usually employed as release for a lock[2].

7. Concurrent Server Process. This ambient construct can be regarded as an abstraction of a multi-threaded server process. It is able to capture and process several ambients having the same name in parallel. In the definition n is a replicated ambient whose each replica is going to capture another ambient called m. Since there is a name restriction quantifier in the scope of the replication sign, which binds the name n, a new, fresh and unique name (denoted by n_k^{uniq}) is generated for each replica of n. One of the consequences of this is that nobody knows from outside the true name of a replica of the ambient n, so each replica of n is inaccessible from outside for anybody (even for another replica of n, too).

5 The Extended Formal Model

In the formal model discussed in this section, we assume that there are some standardized public ambient names, which are known by all contributors. We distinguish the following kinds of public names: addresses (e.g.: $cloud$, $client_1$, ..., $client_n$), message types (e.g.: $reg(istration)$, $request$, $subs(cription)$, $returnValue$, etc.) and parts of some common protocols (e.g.: $lock$, msg, $intf$, $access$, out, o_1, ..., o_s, op). All other ambient names are non-public in the model which follows:

[2] In ambient calculus the capability OPEN $n.P$ is usually used to encode locks [3]. Such a lock can be released with an ambient like $n[Q]$ whose name corresponds with the target ambient of the OPEN capability.

$curAmbProc := root[\ Cloud \mid Client_1 \mid \ldots \mid Client_n\ ]$[3]

In this paper, we focus on the cloud service side and we leave the client side abstract.

5.1 User Actions

In the model user actions are encoded as messages. A user can send the following kinds of messages to the cloud:

$MsgFrame \equiv msg[\ \textsc{In}\ cloud.\textsc{Allow}\ intf.content\ ]$
where *content* can be:

$RegMsg \equiv reg[\ \textsc{Allow}\ CID.\langle UID_x\rangle\ ]$

$SubsMsg \equiv subs[\ \textsc{Allow}\ CID.\langle UID_x, SID_i, pymt\rangle\ ]$

$RequestMsg \equiv request[\ \textsc{In}\ UID_x.\textsc{Allow}\ CID \mid$
$\quad \langle o_i, client_k, args_i\rangle \mid \ldots \mid \langle o_j, client_k, args_j\rangle\ ]$

$AddClMsg \equiv addCl[\ \textsc{In}\ UID_x \mid \textsc{Allow}\ CID.\langle client_k, path_l, UID_{(\text{on } client_l)}\rangle\ ]$

$AddChMsg \equiv addCh[\ \textsc{Allow}\ CID.\langle UID_x, cname\rangle\ ]$

$SubsToChMsg \equiv subsToCh[\ \textsc{Allow}\ CID.\langle UID_x, cname, uname, client_k, pymt\rangle\ ]$

$ShareInfoMsg \equiv share[\ \textsc{In}\ CHID_i \mid \textsc{Allow}\ CID.\langle sndr, rcvr, info\rangle\ ]$

$ShareSvcMsg \equiv share[\ \textsc{In}\ CHID_i \mid \textsc{Allow}\ CID.$
$\quad \langle sndr, rcvr, info, o_i, argsP, argsF\rangle\ ]$

In the definitions above: the ambient *msg* is the frame of a message; the term In *cloud* denotes the address to where the message is sent; the term Allow *intf* allows a (server) mechanism on the target side which uses the public protocol *intf* to capture the message; and the *content* can be various kind of message types. The term Allow *CID* denotes that the messages are sent to a service of a particular cloud which identifies itself with the non-public protocol/credential *CID* (stands for *cloud identifier*).

The first three kinds of messages were introduced in the original model. In a *RegistrationMsg* the user x provides her identifier UID_x that she is going to use in the cloud. By a *SubscriptionMsg* a user subscribes to a cloud service identified by SID_i; the information represented by *pymt* proves that the given user has paid for the service properly.

Again, cloud services provide their functionalities for their environment (users or other services) via actions called service operations in our model. In a *RequestMsg* a user who has subscribed to some services before can request the cloud to perform some service operations belonging to some of these services. Service operation requests are denoted by triples, where o_i and o_j are the unique names of these service operations; $client_k$ is the identifer of a target location (usually a client device) to where the output of a given operation should

[3] The ambient called *root* is a special ambient which is required for the ASM definition of ambient calculus, see [1,15].

be sent by the cloud; and $args_i$ and $args_j$ are the arguments of the corresponding requested service operations. Furthermore, the term In UID_x represents the address of the target user area within the *Cloud*.

The rest of the message types is new in the model. With $AddClMsg$ a user can register a new possible target (client) device or location for the outcomes of the requests initiated by her. Such a message should contain the chosen identifier $client_k$ of the new device, the address $path_l$ of the device and the user identifier $UID_{(\text{on } client_l)}$ used on the given target device.

By $AddChMsg$ users can open new channels, by $SubsToChMsg$ users can subscribe to channels and by $ShareInfoMsg$ and $ShareSvcMsg$ users can share information as well as service operations with some other users registered in the same channel. For the detailed description of the arguments lists of these last four messages, see Sect. 5.3.

5.2 The Cloud Service Architecture

The basic structure of the defined cloud model, which is based on the simplified *Infrastructure as a Service (IaaS)* specification given in [1], is the following:

$Cloud \equiv (\nu\ fw,\ q,\ rescr_1, \ldots rescr_m) cloud[$
$\quad interface\ |$
$\quad fw\ [\ rescr_1[\ service_1\ ]\ | \ldots |\ rescr_l[\ service_1\ ]\ |\ rescr_{l+1}[\ service_2\ ]\ | \ldots |\ rescr_m[$
$service_n\ ]\ |$
$\quad\quad q[\ !\textsc{Open}\ msg\ |\ BasicCloudfunctions\ |\ CTCIfunctions\ |$
$\quad\quad\quad UID_x[userIntf]\ | \ldots |\ UID_y[userIntf]\ |$
$\quad\quad\quad UID_v^{owner}[ownerIntf]\ | \ldots |\ UID_w^{owner}[ownerIntf]\ ]\ ]\ ]$
where
$\quad interface \equiv \textsc{Server}^n_{intf}\ msg.\textsc{In}\ fw.\textsc{In}\ q.n\ \textsc{be}\ msg$

In the cloud definition above, the names of the ambients fw, q and $rescr_1, \ldots rescr_m$ are bound by name restriction. The consequence of this is that the names of these ambients are known only within the cloud service system, and therefore the contents of their body are completely hidden and not accessible at all from outside of the cloud. So each of them can be regarded as an abstraction of a firewall protection.

The ambient expression represented by $interface$ "pulls in" into the area protected by the ambients fw and q any ambient construct which is encompassed by the message frame msg. The purpose of the restricted ambients fw and q is to prevent any malicious content which may cut loose in the body of q after a message frame (msg) has been broken (by Open msg) to leave the cloud together with some sensitive information. For more details we refer to [4].

The restricted ambients $resrc_1, \ldots, resrc_m$, represent computational resources of the cloud. Within each cloud resource some service instances can be deployed. A service may have several deployed instances in a cloud (see instances of $service_1$ in $resrc_1, \ldots, resrc_l$ above).

Every user area is represented by an ambient whose name corresponds to the corresponding user identifier UID_i. Furthermore, the user areas extended with service owner role are denoted by UID_i^{owner}. The terms denoted by $BasicCloudfunctions$ are responsible for cloud user registration and service subscription. Finally the terms denoted by $CTCIfunctions$ encode the client-to-client interaction.

It is beyond the scope of this paper to describe all parts of this model in details (e.g.: the structure of service instances $service_i$, functions of a service owner area $ownerIntf$, the service plots and the ASM agents in $BasicCloudfunctions$). For the specification of these components, we refer to [4].

User Access Layers. A user access layer (or user area) may contain the following mechanisms: accepting user requests and converting them to the format which is compatible with plots[4] ($requestPreprocessor$), accepting new plots (!$\text{A}\small{\text{LLOW}}$ $newPlot$), accepting outputs of service operations (!$\text{A}\small{\text{LLOW}}$ $returnValue$) and some service plots.

$userIntf \equiv$
$\quad requestPreprocessor \mid !\textsc{Allow}\ newPlot \mid clientRegServer \mid !\textsc{Allow}\ returnValue \mid$
$\quad sortingOutput \mid client_1[\ posting_{client_1}\ ] \mid \ldots \mid client_k[\ posting_{client_k}\ ]$
$\quad \textsc{Plot}_{SID_i} \mid \ldots \mid \textsc{Plot}_{SID_j} \mid$
where
$\quad requestPreprocessor \equiv \textsc{Server}^n_{CID}\ request.(o,\ c,\ args).o[\ \textsc{Allow}\ op.\langle c,\ args\rangle\ ]$
$\quad sortingOutput \equiv !(o,\ client,\ a).output[\ \textsc{In}\ client.\textsc{Allow}\ CID \mid \langle o, client, a\rangle\ ]\ ]$
$\quad clientRegServer \equiv \textsc{Server}^n_{CID}\ addCl.(client,\ path,\ UID).(n\ \textsc{Be}\ client \mid$
$\qquad posting_{client}\ )$
$\quad posting_{client_i} \equiv \textsc{Server}^n_{CID}\ output.(o,\ client,\ a).$
$\qquad \textsc{Out}\ client_i.forwardTo_{client_i}.returnValue[\ \textsc{In}\ UID_{(\text{on } client_i)} \mid \langle o, client, a\rangle\ ]\ ]$
$\quad forwardTo_{client_i} \equiv n\ \textsc{Be}\ outgoingMsg.\textsc{Out}\ UID_x.leavingCloud.path_i$
$\quad leavingCloud \equiv \textsc{Out}\ q.\textsc{Out}\ fw.\textsc{Out}\ cloud.outgoingMsg\ \textsc{Be}\ msg$

This paper extends the user areas with some new functionalities. $clientRegServer$ is applied to process every $AddClMsg$ sent by the corresponding user. It creates new communication endpoint for target (client) devices. Each such an endpoint is encoded by an ambient whose name $client_i$ corresponds the given identifier provided in a message $AddClMsg$. By these endpoints outputs of service operations can immediately be directed to registered (client) devices after they are available. Of course, if no target device or a non-registered one is given in a $RequestMsg$, the outcome will be stored in the area of the user.

Every service operation output, which is always delivered within the body of an ambient called $returnValue$, consists of three parts: the name of the performed service operation, the identifier of a target location to where the output should be sent back and the outcome of the performed service operation itself.

$sortingOutput$ distributes every service operation output among the communication endpoints in an ambient called $output$. The mechanism $posting_{client_i}$,

[4] Service plots can accept requests if they are encompassed by ambients whose names are correspond with the unique names of the requested operations ($o_i \ldots o_j$), see the definition of $requestPreprocessor$ above.

which resides in each such a communication endpoint, is responsible to wrap each output of service operations which reaches the corresponding endpoint again into an ambient *returnValue* and to forward it to the specified user $UID_{(\text{on } client_i)}$ on the corresponding device $client_i$.

5.3 Client-to-Client Interaction

Again, the client-to-client interaction in our model is based on the constructs called *channels*. These are represented by ambients with unique names denoted by $CHID_i$ which contain some mechanisms whose purpose is to share some information and service operations among some subscribed users, see below:

```
$CTCIfunctions \equiv$
 $CHID_1[\ channelIntf\ ] \mid \ldots \mid CHID_l[\ channelIntf\ ] \mid$
 $\textsc{Server}^n_{CID}\ addCh.(UID,\ cname).\textsc{ChMgr}(n,\ UID_x,\ cname) \mid$
 $\textsc{Server}^n_{CID}\ subsToCh.(UID, cname, uname, client, pymt).$
   $\textsc{ChSubsMgr}(n,\ UID,\ cname,\ uname,\ client,\ pymt)$
where
 $channelIntf \equiv \textsc{Server}^n_{CID}\ share.(\ (sndr,\ rcvr,\ info).$
   $\langle \mathrm{sndr}, \mathrm{rcvr}, info, \mathbf{undef}, \mathbf{undef}, \mathbf{undef} \rangle \mid$
   $(sndr,\ rcvr,\ info,\ o,\ argsP,\ argsF).$
   $\textsc{SharingMgr}(n,\ sndr,\ rcvr,\ info,\ o,\ argsP,\ argsF)\ )$
```

Every cloud user can create and own some channels by sending the message *AddChMsg* to the cloud, where an instance of the ASM agent ChMgr, which is equipped with a server mechanism, processes such a request and creates a new ambient with unique names for the requested channel, see Sect. 5.3.

If a user would like to subscribe to a channel she should send the message *SubsToChMsg* to the cloud. The server construct belongs to the ASM agent ChSubsMgr is responsible for processing these messages, see Sect. 5.3. In the subscription process the owner of the channel can decide about the rights which can be assigned to a subscribed user. According to the presented high-level model, the employed access rights are encoded by the following static nullary functions: *listening* is a default basic right, because everybody who joins to a channel can receive shared contents; *sending* authorizes a user to send something to only one user at a time; and *broadcasting* permits a user to distribute contents to all member of the channel at once.

Both *ShareInfoMsg* and *ShareSvcMsg* are processed by the same server which belongs to the ASM agent SharingMgr and which is located in the body of each ambient $CHID_i$, see Sect. 5.3. In the case of *ShareInfoMsg* the server first supplements the arguments list of the message with three additional **undef** values, such that it will have the same number of arguments as *ShareSvcMsg* has. Then an instance of the ASM agent SharingMgr can process the *ShareInfoMsg* similarly to *ShareSvcMsg* (the first three arguments are the same for both messages).

Table 3. The ASM agents ChMgr and ChSubsMgr.

```
ChMgr($n$, $UID$, $cname$) $\equiv$
 $ctr\_state : \{InitialState, EndState\}$
 initially $ctr\_state := InitialState$

 if $ctr\_state = InitialState$ then
  $ctr\_state := EndState$
  if $UID \in userIds$ then
   if $ownerOfCh(cname) =$ undef then
    let $CHID =$ new($channelIds$) in
     StoreChannel($CHID$, $cname$, $UID$)
     let $CHConstruct = createChannel( CHID )$
in
     NewAmbientConstruct($CHConstruct$)
where
 $CHConstruct \equiv CHID[\ \underline{\text{Out } n} \mid channelIntf\ ]$
```

```
ChSubsMgr($n$,$UID$,$cname$,$uname$,$client$,$pymt$) $\equiv$
 $ctr\_state : \{InitialState, EndState\}$
 initially $ctr\_state := InitialState$

 if $ctr\_state = InitialState$ then
  $ctr\_state := EndState$
  if $UID \in userIds$ then
   if $ownerOfCh(cname) \neq$ undef then
    if $uname \notin members(cname)$ then
     let $owner = ownerOfCh(cname)$ in
      let $rights =$
           confirmRights($owner$,$uname$,$cname$,$pymt$)
in
       if $rights \neq \emptyset$ then
        StoreMem-
ber($UID$,$uname$,$client$,$cname$,$rights$)
         if $rights \neq \{listening\}$ then
          let $CHID = idOfCh$(cname) in
           NewAmbientConstruct( $returnValue[$
             $\underline{\text{Out } n.\text{In } UID}.\langle cname, client, \text{In } CHID\rangle]$)
```

Establishing a New Channel. ChMgr is a parameterized ASM agent, see in Table 3, which expects *UID* of the cloud user who is going to create a new channel and *cname* which is the name of this channel as arguments. The additional argument n is the unique name of an ambient which was provided by the surrounding server construct and in which the current *AddChMsg* is processed by an instance of this agent (such an argument is also applied in the case of the other ASM agents below).

First the agent checks whether the given *UID* has already been registered on the cloud and whether the given name *cname* has not been used as a name of an existing channel yet (the unary function *ownerOfCh* returns the value **undef** if there is no assigned owner to this name). If it is the case, the agent generates a new and unique identifier denoted by *CHID* for the new channel with the usage of the function **new** which provides a unique and completely fresh element for the given set each time when it is applied. The abstract ASM macro StoreChannel inserts into an abstract database a new entry with all the details of the new channel which are the channel identifier, the channel name and the identifier of the owner.

Then it calls the abstract derived function *createChannel*, which creates an ambient called *CHID* with the terms denoted by *channelIntf* in its body which encode the functions of the new channel. By the abstract tree manipulation operation called NewAmbientConstruct[5] introduced in [1], this generated ambient construct is placed into the ambient tree hierarchy as sibling of the agent.

Although a channel is always created as a sibling of the current instance of ChMgr, but as a first step it leaves the ambient n which was provided by the

[5] This is the only way how an ASM agent can make changes in the ambient tree hierarchy contained by dynamic derived function *curAmbProc* [1].

surrounding server construct and in which the message was processed (see the underlined moving action in *CHConstruct* above). After that it is prepared to serve as a channel for client-to-client interaction (it is supposed that the name *cname* of every channel is somehow announced among the potential users).

Subscribing to a Channel. ChSubsMgr is a parameterized ASM agent, see in Table 3, which expects the following as arguments: *UID* of the user who is going to subscribe to the channel, *cname* which is the name of the channel, *uname* is the name that the user is going to use within the channel, *client* which is the identifier of a registered client device to where the shared content will be forwarded and *pymt* which is some payment details if it is required. A user can register to a channel with different names and various client devices in order to connect these devices via the cloud.

First the agent checks whether the given *UID* and *cname* have already been registered on the cloud and whether the given *uname* has not been used as a name of a member of the channel yet. If it is the case, the agent informs the owner of the channel about the new subscription by applying the abstract ASM macro confirmRights, who responses with a set of access rights to the channel that she composed based on the information given in the subscription.

If the subscription has been accepted by the owner and besides *listening* some other rights are granted to the new user, an ambient construct is created and sent as a message *returnValue* to the user by NewAmbientConstruct. This message contains the capability In *CHID* by which the new user can send messages called *ShareInfoMsg* and *ShareSvcMsg* into the ambient *CHID* which represents the corresponding channel (the owner of a channel also has to subscribe in order to receive this information and to be able to distribute content via the channel).

Sharing Information via a Channel. Every server construct in which the agent SharingMgr is embedded is always located in an ambient which represents a particular channel and whose name corresponds to the identifier of the channel. In order to be able to perform its task, it is required that each instance of SharingMgr knows by some static nullary function called *myChId* the name of the ambient in which it is executed.

SharingMgr is a parameterized ASM agent, see in Table 4, which expects the following arguments: *sndr* is the registered name of the sender, *rcvr* is either the registered name of a receiver or an asterisk "*", *info* is either the content of *ShareInfoMsg* or the description of a shared service operation in *ShareSvcMsg*. The last three arguments are not used in the case of the message *ShareInfoMsg* and the value **undef** is assigned to each of them by the surrounding server construct. In the message *ShareSvcMsg* *o* denotes the unique identifier of the service operation that *sndr* is going to share, *argsP* denotes the arguments of *o* that *rcvr* can freely modify if she calls the operation and *argsF* denotes those part of the argument list of *o*, whose value is fixed by *sndr*.

The agent first generates a new and unique operation identifier for the service operation *o* in the control state *InitialState*. This new identifier which is

Table 4. The ASM agent SharingMgr.

```
SharingMgr($n$, $sndr$, $rcvr$, $info$, $o$, $argsP$, $argsF$) $\equiv$
 $ctr\_state : \{InitialState, SharingState, EndState\}$
 initially $ctr\_state := InitialState$
 if $ctr\_state = InitialState$ then
   $ctr\_state := SharingState$
   if $o \neq$ undef then                    //svc. sharing
     let $newOpId$ = new($sharedOpIds$) in
       $shOp = newOpId$
   else $shOp$ = undef                       //msg. sharing
 if $ctr\_state = SharingState$ then
   $ctr\_state := EndState$
   let $cname = getChannelName(myChId)$ in
     if $sndr \in members(cname)$ then
       let $rights = getRights(cname, sndr)$ in
         if $rcvr$ = "*" then                //broadcasting a msg.
           if $boradcasting \in rights$ then
             forall $M \in members(cname)$ do
               let $UID$ = getId($M$), $client$ = getAddress($M$) in
                 if $shOp$ = undef then
                   NewAmbientConstruct( $sharedM_{content_1}$ )
                 else
                   NewAmbientConstruct( $sharedM_{content_2}$ )
                 let $UID_{sndr}$ = getId($sndr$) in
                   NewAmbientConstruct( $sharedPlot$ )
         else                                //sending a msg.
           if $sending \in rights$ and $rcvr \in members(cname)$
           then
             let $UID$= getId($rcvr$), $client$= getAddress($rcvr$) in
               if $shOp$ = undef then
                 NewAmbientConstruct( $sharedM_{content_1}$ )
               else
                 NewAmbientConstruct( $sharedM_{content_2}$ )
               let $UID_{sndr}$ = getId($sndr$) in
                 NewAmbientConstruct( $sharedPlot$ )
where
  $sharedM_{content_i} \equiv returnValue[$
    $\underline{\text{Out } n.\text{Out } myChId.\text{In } UID}.\langle cname, client, content_i \rangle$
  $]$
  $content_1 \equiv \{$"sender:" $sndr$, "content:" $info\}$
  $content_2 \equiv \{$"sender:" $sndr$, "operation:" $shOp$,
    "arguments:" $argsP$, "description:" $info\}$
  $sharedPlot \equiv$
    $newPlot[\ \underline{\text{Out } n.\text{Out } myChId.\text{In } UID} \mid \text{Plot}_{shOp}\ ]$
  $\text{Plot}_{shOp} \equiv \text{Server}^{s}_{op}\ shOp.trigger_o$
  $trigger_o \equiv (\nu\ tmp)$
    $(client, argsP).(\text{Out } UID.\text{In } UID_{sndr}.s \text{ Be } request \mid$
    $\text{Allow } CID.\langle o, tmp, (argsP \setminus argsF) + argsF \rangle \mid$
    $tmp[\ \text{Allow } output \mid CID[\ (o, c, a).\text{Out } UID_{sndr}.$
      $\text{In } UID.tmp \text{ Be } returnValue.\langle shOp, client, a \rangle\ ]\ ]\ )$
```

stored in the nullary location function *shOp* will be announced to the channel member(s) specified in *rcvr*. In the control state *SharingState* the agent checks whether the *sndr* is a registered member of the channel by calling the function *members*(*cname*). Then if the given value of *rcvr* is equal to "*" the agent broadcasts the content of the current message to all members of the channel, see code branch bordered by the first rectangular frame below. Otherwise if the value of *rcvr* corresponds to the name of a particular member of the channel, the agent sends the content of the current message only to her, see the code branch bordered by the second rectangular frame below.

Apart from the number of users to whom the information is sent the both code branches mentioned above define the same actions. Accordingly at the end of the processing of *ShareInfoMsg* the agent sends to the member(s) specified in *rcvr* the message $sharedM_{content_1}$, which contains the sender *sndr* and the shared information $info$.

At the end of the processing of *ShareSvcMsg* two ambient constructs are created by NewAmbientConstruct. The first one is the message $sharedM_{content_2}$ and it is sent to the member(s) specified in *rcvr*. It contains the sender *sndr*, the new operation identifier *shOp*, the list of public arguments *argsP* and the informal description of the shared operation denoted by $info$.

The second ambient construct is the plot PLOT_{shOp} enclosed by the ambient *newPlot* and equipped with some additional ambient actions (see the underlined capabilities in the definition of *sharedPlot*) which move the entire construct into the user area of the channel member(s) specified in *rcvr*, where the plot will be accepted by the term !ALLOW *newPlot*.

The execution of the shared service operation *shOp* can be requested in a usual *RequestMsg* as normal service operations. The PLOT_{shOp} is a plot, which can accept service operation requests for *shOp* several times. It is special plot, because instead of triggering the execution of *shOp* as in the case of a normal operation a normal plot does, see [4], it converts the original request to another request for operation *o* by applying the term $trigger_o$. This means that it substitutes the operation identifier *o* for *shOp*, it completes its arguments list with *argsF* and it forwards the request for *o* to the user area of the user *sndr* who actually has right to trigger the execution of the operation *o*.

To the new request the name restricted ambient *tmp* is attached, whose purpose is similar to the communication endpoints of registered clients. Namely, it is placed into the user area of *sndr* temporary and it is responsible for forwarding the outcome of this particular request from the user area of *sndr* to the user area of the user who initiated the request. It is beyond the scope of this chapter to present a reduction how a particular request for a shared operation is processed in our model, but we refer to [18] for more details.

6 Anonymous Docking Service

If we apply the scenario proposed in Sect. 2 and depicted on Fig. 1b, according to which we shift (among others) the client-to-client functionality to client side and wrap into a middleware, then no traces of the user activities belonging to the shared services will be left on the cloud, since all the service operations which are shared via a channel are used on behalf of its initial distributor.

Many scenario can make a profit on this fact, which require some anonymously usable cloud services. For instance, one of the possible use cases arises in a *multi-clouds* approach which enables many-to-many relationship between cloud service providers and customers of the middleware, such that the middleware architecture is capable to treat intermediate results exchanged among the requested cloud services. It may become necessary to store intermediate results on a third party cloud exploiting infrastructure as a service, and to ensure that after completion of the temporary use of this *docking service* no trace of the customers is left.

In the case study discussed in this section, we introduce a new kind of requests called *pipelined requests*, which can be composed from some normal service operation requests such that the requested services are able to exchange data according to a predefined information flow pattern. Below we also extend our formal model to be able to process this new kind of requests in a distributed way and to be able to anonymously store the intermediate results exchanged among some requested services on (probably) a third-party IaaS.

In our approach we assume that the middleware mentioned above (or its provider) has access to such a third-party storage service, whose operations are shared with all users of the middleware via some kind of public channel. Since these users access to the third-party docking service on behalf of the middleware (provider), their personal data is not given/forwarded to any third-party for any service subscription.

A complex pipelined request can be regarded as an extension of *RequestMsg* defined in Sect. 5.1, see an example below:

$$\begin{aligned}&RequestMsg_{pipelined} \equiv request[\ \textsc{In}\ UID_x.\textsc{Allow}\ CID \mid \langle P_1\rangle \mid \langle P_2\rangle \mid \\ &\quad \langle o_i, P_1, args_i\rangle \mid \langle o_j, P_2, args_j\rangle \mid \langle o_k, client, \{arg_1, \ldots, P_1, \ldots, P_2, \ldots, arg_n\}\rangle\ ]\end{aligned}$$

$RequestMsg_{pipelined}$ also contains the triples which denote the usual service requests, but it can also contains some singletons which declare the names of some information flow (or pipe) denoted by $P_1, \ldots, P_n$. If such a pipe name appears as the target location of the output of a requested operation (see the request triples for o_i and o_j above), then this output should be stored on a docking service, instead of sending to the user who initiated the request.

If the name of some pipes appears in the argument list of some operation requests, then the execution of these requests is blocked as long as all the inputs provided via the mentioned pipes will be available. Every pipe always describes a one-to-one or a one-to-many relationships (one operation can provide data to many) and it is always local to its containing *RequestMsg*.

6.1 New Assumptions and Changes in the Model

Now, it is assumed that each user of the middleware has access to the following two shared service operations which were distributed on behalf of the middleware provider via some public channel after each user registration:

- *sharedStore* is a shared version of a service operation whose task is to store some data in a filesystem on a third-party IaaS. It has two arguments, which are freely modifiable by the users. The first is an identifier (a pipe name) and the second is the data which are going to be stored.
- *sharedReceive* is a shared version of another service operation which belongs to the same third party IaaS as *sharedStore*. It has only one freely modifiable argument, the identifier by which some stored data can be retrieved. If no data is stored with the given identifier, the operation blocks until some data bound to such an identifier appear on the third-party IaaS.

In order to adapt the model to the new pipelined requests only the ambient expression represented by *requestPreprocessor* has to be replaced which was given as a part of the definition of user areas in Sect. 5.2:

$$\begin{aligned}&requestPreprocessor \equiv \textsc{Server}^{n}_{CID}\ request.(\\ &\quad !(p).\textsc{Listener}_{pipe}(p) \mid !(o,\ c,\ args).\textsc{Listener}_{req}(o,\ c,\ args) \mid \textsc{RequestMgr}(\mathrm{n})\ )\end{aligned}$$

Table 5. The ASM agents $\textsc{Listener}_{pipe}$ and $\textsc{Listener}_{req}$.

```
$\textsc{Listener}_{pipe}(p) \equiv$
  $ctr\_state : \{InitialState, EndState\}$
  initially $ctr\_state := InitialState$
  if $ctr\_state = InitialState$ then
    $ctr\_state := EndState$
    $\textsc{Add}(p, mailbox_{pipe})$
```

```
$\textsc{Listener}_{req}(o, c, args) \equiv$
  $ctr\_state : \{InitialState, EndState\}$
  initially $ctr\_state := InitialState$
  if $ctr\_state = InitialState$ then
    $ctr\_state := EndState$
    $\textsc{Add}(\langle o,\ c,\ args\rangle, mailbox_{req})$
```

The new expression is an ambient server construct that is able to capture (both normal and pipelined) service operation requests arriving at a user area and able to prepare them for execution with the help of the three ASM agents called $\textsc{Listener}_{pipe}$, $\textsc{Listener}_{req}$ and $\textsc{RequestMgr}$(n).

$\textsc{Listener}_{pipe}$ and $\textsc{Listener}_{req}$ are very simple parameterized ASM agents, see in Table 5, whose several instances are available in the server construct referred by *requestPreprocessor*. Each replica of $\textsc{Listener}_{pipe}$ can capture a singleton containing a pipe name and mediates it to the agent $\textsc{RequestMgr}$ via the shared dynamic function $mailbox_{pipe}$[6]. Replicas of $\textsc{Listener}_{req}$ can capture request triples, respectively, and also forward them to the agent $\textsc{RequestMgr}$ via the shared dynamic function $mailbox_{req}$.

6.2 Request Preprocessing

$\textsc{RequestMgr}$ is a parameterized ASM agent, see in Table 6, whose only argument is n which is the unique name of an ambient provided by the surrounding server construct and in which the content of current *RequestMsg* is preprocessed by an instance of this agent.

In the control state *InitialState* the agent first waits until every singleton and every triple in n are captured by $\textsc{Listener}_{pipe}$ and $\textsc{Listener}_{req}$. Then in the control state *PreProcessing* all request triples contained by the captured message will be prepared for execution in parallel.

In the next step, each request triple $\langle o,\ c,\ args\rangle$ is checked whether its execution is independent from other requests or in other words none of the pipe names occurs in *args*. If it is the case, the agent also checks whether the target location c does not correspond with any pipe name. If this is true as well, then the current request is a normal request which is not connected to any pipe, so it is simply converted into a service plot compatible format as before with $\textsc{NewAmbientConstruct}$ (see $request(n,\ o,\ c,\ args)$ in Table 6).

In that case if a pipe denoted by P_{out} is specified as a target location in the request, the agent generates a new and unique global identifier denoted by PID for the pipe with the function **new**. PID substitutes for P_{out} in the request

[6] In our applied ambient ASM-based formal method, ASM agents can communicate with each other directly via shared functions if and only if they are sibling of each other [1].

Table 6. The ASM agent RequestMgr.

```
RequestMgr($n$) $\equiv$
 $ctr\_state : \{InitialState, PreprocessingState, EndState\}$
 initially $ctr\_state := InitialState$
 if $ctr\_state = InitialState$ then
  if all async outputs are captured in $n$ then
   $ctr\_state := PreprocessingState$
 if $ctr\_state = PreprocessingState$ then
   $ctr\_state := EndState$
   forall $R = \langle o,\ c,\ args\rangle \in mailbox_{req}$ do
    if $R \in independentReq(mailbox_{req},\ mailbox_{pipe})$ then
     if forall $P_{out} \in mailbox_{pipe}$ $P_{out} \neq c$ then
      NewAmbientConstruct( $request(n, o, c, args)$ )
     else
      choose $P_{out} \in mailbox_{pipe}$ with $P_{out} = c$ do
       let $PID$ = new($pipeIds$) in
        NewAmbientConstruct(
         $request(n, o, PID, args)$ | $tmpEndPoint(PID, P_{out})$ )
    else
     let $RID$ = new($requestIds$) in
      forall $P_{in} \in mailbox_{pipe}$ with $P_{in} \in args$ do
       NewAmbientConstruct( $receiveMsg(RID, P_{in})$ )
      if forall $P_{out} \in mailbox_{pipe}$ $P_{out} \neq c$ then
       NewAmbientConstruct(
        $blockedRequest($ $request(RID, o, c, args)$ ) )
      else
       choose $P_{out} \in mailbox_{pipe}$ with $P_{out} = c$ do
        let $PID$ = new($pipeIds$) in
         NewAmbientConstruct(
          $blockedRequest($ $request(RID, o, PID, args)$ ) |
          $tmpEndPoint(PID, P_{out})$ )
where
 all async outputs are captured in $n \equiv$ forall $o, c, args, P, Q$
  $(ambBody(n) \neq \langle o,\ c,\ args\rangle \mid Q \wedge ambBody(n) \neq \langle P\rangle \mid Q)$
 $independentReq(X,\ Y) \equiv$
  $\{\ \langle o,\ c,\ args\rangle \in X \mid$ forall $P_{in} \in Y\ (P_{in} \notin args)$
 $\}$
 $request(amb,\ o,\ target,\ args) \equiv$
  $o$[ Out $amb$.Allow $op.\langle target,\ args\rangle$ ]
 $tmpEndPoint(PID, P_{out}) \equiv$
  $PID$[ Out $n$.Allow $output$ | $CID$[ $(o, c, a).sharedStore$[
    Out $PID$.Allow $op.\langle\ c, \{n{:}P_{out}, a\}\rangle$ ] ] ]
 $receiveMsg(RID, P_{in}) \equiv$
  $sharedReceive$[ Out $n$.Allow $op.\langle\ RID, n{:}P_{in}\rangle$ ]
 $blockedReq(request) \equiv$
  $RID$[Out $n$ | !Allow $output.CID$[ ] |
   !$(o, c, a)$.Listener$_{output}(o, c, a)$ |
   ReqTrigger($n$, $mailbox_{pipe}$, $request$) ]
```

triple as the target location of the output. When the modified ambient request is created with NewAmbientConstruct another ambient term denoted by *tmpEndPoint* attached to it, whose purpose is similar to the communication endpoints of the registered clients. Namely, it refers to an ambient called PID, so the output of the service operation eventually arrives at the body of this ambient. The aim of the mechanism located in the body of the ambient PID is to trigger the shared operation *sharedStore* which will store the output bound to the global pipe identifier $n{:}P_{out}$ on the third-party IaaS (P_{out} alone cannot be applied as a unique global identifier of the pipe on a third-party storage, since it is always given by a user; hence, it should be extended with n as prefix, because n always refer to a unique name in the case of each captured *RequestMsg*).

In that case if some of the pipe names occur in *args* (non-independent request), the request must be blocked until all the inputs referred by these pipes are available. First a unique identifier denoted by RID is generated for the request, which is applied as the name of the ambient, into where the request is enclosed and at where the required inputs from the third-party IaaS arrive eventually. Then a request for shared service operation *sharedReceive* is triggered in parallel for each such a pipe with the argument $n{:}P_{in}$ and with the target location RID (see $receiveMsg(RID,\ P_{in})$ in Table 6). These requests

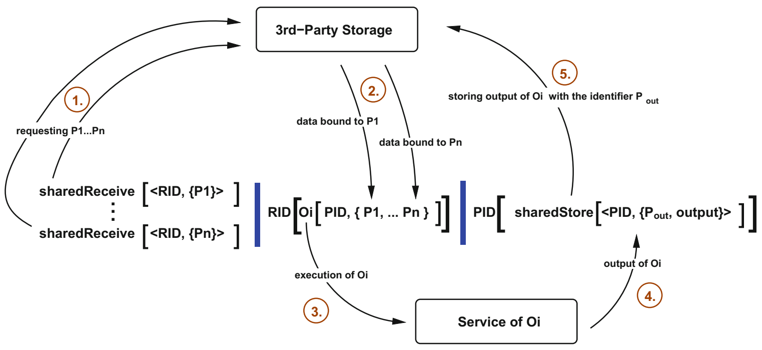
3rd–Party Storage
1.
requesting P1...Pn
2.
data bound to P1
data bound to Pn
5.
storing output of Oi with the identifier P out
sharedReceive [<RID, {P1}>]
sharedReceive [<RID, {Pn}>]
RID [Oi [PID, { P1, ... Pn }]]
PID [sharedStore [<PID, {Pout, output}>]]
execution of Oi
3.
output of Oi
4.
Service of Oi

Fig. 2. Execution of pipelined service operation o_i.

block until some data bound to the global pipe identifer $n{:}P_{in}$ is not available on the third-party IaaS.

Concurrently with the sending of *sharedReceive* messages, an ambient construct called *blockedRequest* is created, which denotes the ambient RID and some ASM agents in its body. The ambient RID serves as the target location of the outputs of the triggered *sharedReceive* requests. This ambient also contains several replicas of the abstract ASM agent LISTENER$_{output}$ and one instance of the ASM agent REQTRIGGER. Each LISTENER$_{output}$ is responsible for capturing an output triple of an executed *sharedReceive* and for delivering it to REQTRIGGER via a shared dynamic function. REQTRIGGER is also an abstract ASM agent whose task is to add all the expected inputs provided by other services via pipes to the argument list of the current request and to then trigger this request as it is specified in the agent REQTRIGGER's argument list. If the target location of a non-independent request is also a pipe, then the original target location of this request is replaced with PID and *tmpEndPoint* is attached to *tmpEndPoint* like in the previous case above.

Figure 2 depicts a generalized summary how a request which is part of a pipelined *RequestMsg* is processed in the model. According to it, if the request for the operation o_i requires inputs from other services, some *sharedReceive* requests are sent to the third-party IaaS on behalf of the middleware and the request for o_i is blocked in an ambient whose name is denoted by RID. After all the necessary data have arrived at the ambient RID and they have been added to the argument list, the request for o_i is triggered and executed. If the given target location of the output of o_i refers to another pipe, then this output is delivered into the ambient PID instead of a client. From here the output is forwarded in a *sharedStore* request message and stored on the third-party IaaS on behalf of the middleware as well.

7 Conclusions

In this paper we extended our formerly given cloud model with the high-level formal definitions of some client-to-client interaction functions, by which not only information, but cloud service functions can be also shared among the cloud users. Our approach is general enough to manage situation in which a user who has access to a shared service operation to share it again with some other users via a channel (who in turn may share it again, etc.).

Furthermore, if we apply the scenario proposed in Sect. 2 and depicted on Fig. 1b, according to which we shift (among others) the client-to-client functionality to client side and wrap into a middleware, then no traces of the user activities belonging to the shared services will be left on the cloud, since all the service operations which are shared via a channel are used on behalf of its initial distributor. As it was showed this consideration can facilitate the development of anonymously accessible cloud services. The consequence of this is that if a cloud user who has contracts with some service providers completely or partially shares some services via a channel, then she should be aware of the fact that all generated costs caused by the usage of these shared services will be allocated to her.

The specification described in Sect. 6 can lead to a solution of some problems regarding nowadays *(web) mashup* services, too. A mashup is a composed application, using elements from different sources. Namely, the examination of the security requirements for mashups [19] demands among others stronger separation guarantees between the executable components, but at the same time also require the possibility of interaction between these separated components. According to our opinion the formal specification defined above for complex pipelined requests can also be a good basis for overcoming these two problems of mashup services.

Acknowledgements. This research has been supported by the Christian Doppler Society.

References

1. Bósa, K.: Formal modeling of mobile computing systems based on ambient abstract state machines. In: Schewe, K.-D., Thalheim, B. (eds.) SDKB 2013. LNCS, vol. 7693, pp. 18–49. Springer, Heidelberg (2013)
2. Börger, E., Stark, R.F.: Abstract State Machines: A Method for High-Level System Design and Analysis. Springer, Secaucus (2003)
3. Cardelli, L., Gordon, A.D.: Mobile ambients. Theor. Comput. Sci. **240**, 177–213 (2000)
4. Bósa, K.: An ambient ASM model for cloud architectures. Formal Aspects of Computing (2013, submitted)
5. Ma, H., Schewe, K.-D., Thalheim, B., Wang, Q.: Abstract state services. In: Song, I.-Y., et al. (eds.) ER Workshops 2008. LNCS, vol. 5232, pp. 406–415. Springer, Heidelberg (2008)
6. Ma, H., Schewe, K.D., Thalheim, B., Wang, Q.: A theory of data-intensive software services. Serv. Orient. Comput. Appl. **3**, 263–283 (2009)
7. Kozen, D.: Kleene algebra with tests. Trans. Program. Lang. Syst. **19**, 427–443 (1997)

8. Boudol, G., Castellani, I., Hennessy, M., Kiehn, A.: A theory of processes with localities. Formal Aspects Comput. **6**, 165–200 (1994). doi:10.1007/BF01221098
9. Cardelli, L.: Mobility and security. In: Bauer, F.L., Steinbrüggen, R., (eds.) Proceedings of NATO Advanced Study Institute on Foundations of Secure Computation. Lecture Notes for Marktoberdorf Summer School 1999 (A Summary of Several Ambient Calculus Papers), pp. 3–37. IOS Press (1999)
10. Schewe, K.D., Thalheim, B.: Personalisation of web information systems - a term rewriting approach. Data Knowl. Eng. **62**, 101–117 (2007)
11. Tanaka, Y.: Meme Media and Meme Market Architectures: Knowledge Media for Editing, Distributing, and Managing Intellectual Resources. Wiley, New York (2003)
12. Ma, H., Schewe, K.D., Thalheim, B., Wang, Q.: A formal model for the interoperability of service clouds. Serv. Orient. Comput. Appl. **6**, 189–205 (2012)
13. Jarraya, Y., Eghtesadi, A., Debbabi, M., Zhang, Y., Pourzandi, M.: Cloud calculus: security verification in elastic cloud computing platform. In: Smari, W.W., Fox, G.C. (eds.) CTS, pp. 447–454. IEEE (2012)
14. Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes, Parts I. and II. Inf. Comput. **100**, 1–77 (1992)
15. Börger, E., Cisternino, A., Gervasi, V.: Ambient abstract state machines with applications. J. CSS (Special Issue in Honor of Amir Pnueli) **78**, 939–959 (2012)
16. Valente, M., Bigonha, R., Loureiro, A., Maia, M.: Abstractions for mobile computation in ASM. In: Graham, P., Maheswaran, M. (eds.) Proceedings of the International Conference on Internet Computing, IC 2000, Las Vegas, Nevada, USA, 26–29 June 2000, pp. 165–172. CSREA Press (2000)
17. Bósa, K.: A formal model of a cloud service architecture in terms of ambient ASM. Technical report, Christian Doppler Laboratory for Client-Centric Cloud Computing (CDCC), Johannes Kepler University Linz, Austria (2012)
18. Bósa, K.: An ambient ASM model for client-to-client interaction via cloud computing. In: Proceedings of the 8th International Conference on Software and Data Technologies (ICSOFT), Reykjavik, Iceland, pp. 459–470 (Best Paper Award). SciTePress (2013)
19. De Ryck, P., Decat, M., Desmet, L., Piessens, F., Joosen, W.: Security of web mashups: a survey. In: Aura, T., Järvinen, K., Nyberg, K. (eds.) NordSec 2010. LNCS, vol. 7127, pp. 223–238. Springer, Heidelberg (2012)