Injecting Distribution in Casl

Matteo Dell'Amico and Maura Cerioli[1]

DISI–Dipartimento di Informatica e Scienze dell'Informazione
Università di Genova, Via Dodecaneso, 35, 16146 Genova, Italy
cerioli@disi.unige.it,dellamico@disi.unige.it

Abstract. In this paper we present a first attempt at the development of a library in the Casl-Ltl specification language providing primitives to represent connectivity and communication in a distributed system.
The focus, in particular, is on peer-to-peer, which presents more challanges than the client-server paradigm, because of the higher degree of anarchy and the large amount of middleware providing similar, though quite different, features in support of it.
From our experience on the definition of this library, we draw some methodological lessons on how to deal with the capture of complex software systems, as opposite to classical libraries representing standard or mathematical datatypes.

Introduction

The mechanism of libraries is a powerful way of providing extensions of a language, when it is not required to add new concepts to the language semantics, but only to have a richer interface to the same semantics. In particular, it is largely used in programming languages to support the programmer productivity by providing solutions to the most common tasks and abstractions of low-level operations, often going by the collective name of middleware.

We advocate the very same approach to specifications. That is, specification languages to be effective in the process of software development should provide libraries not only for the standard datatypes like integers, lists and sets, but also for middleware primitives. Indeed, the development of systems nowadays relies on (and is influenced by) the middleware they will run on. Thus, on one side the developers must accommodate the other subsystems around those given by the platform, and hence the middleware or, better, some abstraction of the middleware has to be taken into account from the very beginning of the development process. But, on the other side, the middleware has to be *used*, not to be *developed*, and hence it is not responsibility of the developers, who should not be burdened with its axiomatization in the first steps of development. Therefore, specification languages should provide the abstraction of middleware, and, mimicking the programming language world, the most natural approach to the representation of middleware is the definition of a library.

One of the most widely used middleware is that for distribution. Indeed, most applications nowadays are distributed and the management of low-level

Dagstuhl Seminar Proceedings 05081
Foundations of Global Computing
http://drops.dagstuhl.de/opus/volltexte/2006/298

protocols is usually left to the run-time environment. Thus, in this paper we will focalize on the building of a library for distribution in CASL.

While the client/server paradigm is well established, the newly emerging peer-to-peer (P2P) one is less stable. There are several platforms designed to support it, each one offering somewhat different sets of operations. On the other side, client/server is just a special case of P2P. Thus, it seems more productive to focus on P2P and derive other paradigm of distribution by specialization.

From the careful analysis of many middleware for the P2P, we have produced a hierarchy of specifications providing an abstract description of peers and nets at different levels of connection. Such specifications form a first kernel of a library for P2P middleware. It is worth to note that the process of definition of this library is quite different from that for standard data types. Indeed, in the standard cases, like for instance integers, reals or several kinds of collections, the type to be described is well known and formally defined in some mathematical language. Thus, the task of the specifier is to translate the given definition, or at least its most relevant properties, in terms of the specification language. Therefore, in the standard cases, building a specification library follows a, so to speak, *platonic* approach: the object to be described is an idea in the *hyperuranium* and the specifier just has to capture its shadow as precisely as possible.

In the P2P case, on the contrary, the first step is understanding what has to be described. Indeed, there is not just one idea of *P2P middleware*, nor there is a formal definition of any proposal. Thus, one of the lessons learned from this attempt at a P2P library, is a *method* to be followed in order to define libraries for technological entities. The naive approach of analyzing the existing middlewares and giving a specification including all features does not work, because not only different middlewares may approach the same problem in contradictory ways, but especially as the class of software supporting P2P is quite large (from low-level protocol implementation to applications), and including all the possible features in one library would create an unusable monster. We found most useful a problem-driven approach. That is, we started from the features needed by some applications based on a P2P architecture. Then, we analyzed the middlewares providing them, the exact form in which they were realized and if they were usually associated to other services. Finally, from this collection of concrete example we abstracted and specified the result of our abstraction.

We also found out during the definition of this library that there are schemas of specifications which present themselves in several cases. Thus, we also propose a bit of syntactic sugar to simplify this kind of specifications.

Paper structure Sec. 1 introduces the preliminaries about CASL and P2P, Sec. 2 describes the style of specifications adopted and some syntactic sugar, and finally Sec. 3 sketches the part of the specification library used for a toy application; the other specifications of the library are collected in an Appendix.

1 The Context of the Work

1.1 Casl and Casl-Ltl

The algebraic specification language CASL has been developed as central part of the `CoFI` initiative[1]. It provides constructs for declaring basic and structured specifications, whose semantics is a class of *partial* first-order models, and architectural specifications, whose semantics is, roughly speaking, a (higher-order) function over the semantics of basic specifications. Thus, the natural semantics of CASL specifications is the *loose* one: all the partial first-order structures satisfying its axioms are models of a basic specification. However, the models may be restricted to the initial (free) ones, by means of a structuring construct, so that methods based on initial semantics may be accommodate as well.

The building blocks of basic specifications are declarations of (sub)sorts, operations and predicates, giving a signature, and axioms on that signature. Operations may be total or *partial*, denoted by a question mark in the arity. CASL also accommodates subsorting; but, here we do not explicitely use it.

The structuring operators are the usual in algebraic specification languages, like union, (free) extension, renaming and hiding. We will use mostly union, extension and generic specifications. The latter being less standard, let us discuss a bit its semantics and usage. A generic specification is named and consists of

- a list of *formal parameters*, which are place holder specifications to be replaced, in the instantiation phase, by more detailed specifications, the actual parameter, possibly using a *fitting morphism* to connect the symbols used in the formal parameters to those in the actual parameters;
- a list of *imports*, which are specifications to be used as they are, for instance that of integer numbers;
- a *body* specification, describing the features to be added to the parameters and the imports by the specification.

The result of an instantion is, roughly speaking, the enrichment of the (union of) the actual parameters and the imports by (the translation of) the body (by the fitting morphisms).

For a complete description of CASL, we refer to [5].

Casl-Ltl and Generalized Labeled Transition Systems It is important to note that CASL is one of a *family* of languages, sharing common constructs and their semantics. For instance, there are restrictions of CASL without partial functions, and/or subsorting, and/or predicates, so that the resulting language may be translated in other less rich languages in order to use tools built for such languages. On the converse, there are extensions of CASL by constructs and corresponding semantics to deal with specific problem. For instance, there is higher-order CASL (see e.g. [11]) and state-based CASL (see e.g. [1]).

---

[1] See the site `http://www.brics.dk/Projects/CoFI`.

In the sequel we will use CASL-LTL (see [8]), which is designed to describe *generalized labeled transition systems* (glts from now on).

A glts may be used to represent the evolution of a dynamic system; it consists of a set of *states* of the system, one of *labels*, one of *information* and finally the *transition relation*, representing the evolution capabilities of the system. Any element of the transition relation is a tuple consisting of the starting and the final states, a label, capturing all the data about the transition which are relevant to the external world, and an information, capturing all the data about the transition which are relevant only to the system itself. For instance, if a system is keeping track of the number of sent messages, the transition corresponding to sending all the messages in a queue will have the message list coded in the label and the number of sent messages in the info part, to be used to update the internal counter. Any state of the system corresponds to the process having an evolution tree determined by the transition system itself, where each branch is given by a transition of the system and represents a *capability* of moving of the parent state.

A glts may be specified by using CASL-LTL. Indeed, CASL-LTL allows to declare dynamic sorts, by **dsort** *ds* **label** *l_ds* **info** *i_ds*. This CASL-LTL construct semantically corresponds to the declaration of the sorts *ds*, *l_ds*, and *i_ds* for the states, the labels and the information of the glts, and of the transition predicate **preds** $\_\_ : \_\_ \xrightarrow{\_\_} \_\_ : i\_ds \times ds \times l\_ds \times ds$, as well.

Thus, each element s of sort ds in a model M (an algebra or first-order structure) of the above specification corresponds to a process modelled by a transition tree with initial state s determined by the glts $(i\_ds^M, ds^M, l\_ds^M, \_\_ : \_\_ \xrightarrow{\_\_} \_\_^M)$[2].

The most important extension of CASL-LTL w.r.t. CASL is the enrichment of the logic by constructs from a branching-time CTL-style temporal logic, which effectively increase the expressive power of the language.

In the sequel we will use an obvious shortcut for dynamic specifications with an irrelevant information or label part, that is we will drop any reference to the immaterial aspect. The general case is computed from the shorter version, by adding a sort with just one element for the missing component and decorating all the transitions by that element too.

1.2 Distributed Systems and P2P: State of the Art

The "*peer-to-peer*" term is widely used with quite a fuzzy meaning. [7] provides, along with a survey of many P2P applications, some informal definitions of the term. At the core of it we can anyway find the common concept of *decentralization*. We can break it down in three independent – yet anyway related – components: decentralization of overlay network architecture, of location of valuable resources and of content production. There is a certain degree of sinergy between these aspects, and many application embrace more than one of them. In this paper, we focus on software with a decentralized network topology. We

[2] Given a Σ algebra A, and a sort s of Σ, s^A denotes the interpretation of s in A; similarly for the operation and predicates of Σ.

argue that a project choice of this kind can favour a good implementation of the other two objectives.

Decentralization in Network Architecture As opposed to the client-server paradigm, in which there is a clear distinction of roles between the actors and complexity is reduced by offering all services only in a central node (the server), peers can (and do) both offer to and use services provided by other nodes. A sensibly projected application can benefit from decentralization by having a notably good fault-tolerance (P2P applications normally have no single point of failure). On the flip side, applications become more complex. Moreover, security becomes more difficult to handle due to the fact that a peer has to communicate with many untrusted nodes. The use of middleware can be a good choice because it can solve various common problems, effectively hiding complexity to the application developer.

Decentralization in Location of Resources Using distributed resources such as processing power, bandwidth or storage space can prove itself to be a cost-effective strategy, since facilities located at the edge of the network are usually cheap and often unused. Of course, use of decentralized resources arises naturally in applications with a decentralized architecture. Efficiently scheduling the use of resources is an important issue, which again can be effectively tackled by middleware.

Decentralization of Content Production An important social implication of decentralized applications is that the distinction between publishers and recipients of information tends to fade away, since it often happens that publishing new information becomes just as easy as accessing information submitted by others. This is often not the case with other, more traditional, kinds of distributed applications (the WWW is a prominent example). Another side-effect of decentralized applications is that it becomes more and more technically unfeasible to impose a central control or filter on the produced content. Moreover, various applications provide a degree of anonimity to the user, protecting him from the menace of retaliation.

Characteristics of P2P applications have been successful in various different areas. In the following, we will highlight some examples.

- Scalability and efficience make P2P a good choice for content-distribution networks where performance is an issue: file-sharing is obviously the killer application. Other interesting fields are tools for sharing bandwith, lowering cost and increasing efficience for big uploads, with [2] or without [6] realtime constraints that can be useful for multimedia streaming.
- Decentralization on the network makes anonimity more feasible, due to the absence of a privileged observing position. Freenet[4] is a general-purpose network created on top of the Internet with the task of preserving anonimity and avoiding censorship.

– Regardless of their actual underlying network connections[3], Instant Messaging applications (such as Jabber [10]) implement the concept of peers directly communicating with each other.
– The problem of finding information in a great decentralized network can be addressed by using solutions such as distributed hash tables (Chord [12] is a notably simple, yet efficient, one).

2 Specification Style for the P2P Library

We aim at the definition of a library for the abstract description of P2P middleware, in an extremely loose style, expecting each specification to have several interesting and concrete models: the implementations by different middlewares. Thus, we adopt an observational style, in the sense that we introduce the sorts we need to categorize the objects we will be working with and functions and predicates to extract from the elements the values of some of their aspects, which we regard as relevant for the applications to be built upon our infrastructure. However, we are not relying on the observers to distinguish elements, as in most observational approaches. Indeed, by the nature of our library, the observer set is continuously extended as new aspects of the nodes and nets are introduced by the library specifiers and end-users. Thus, the fact that the current set of observers cannot distinguish between two elements is not a clue of their equality, it could as well be an indication of some aspect still to be taken into account. Therefore, our approach has in common with more traditional observational approaches (e.g., the pioneering [9], we defer to [3] for further references) only the intuition of the black-box approach and the use of the word observer.

For instance let us consider the case of the most basic specification in our library, the one of *peer*. A *peer* is the abstraction of any node in a net. It has a persistent identity, the capability to connect to a net using a given address and to disconnet from the net. Thus, we leave underspecified how elements of the sort peer are made and introduce functions extracting the identity, address, and online status from such elements, as in the following signature[4]

sort *PeerId*
dsort *Peer* **label** *PeerL*
ops *online* : *Address* →? *PeerL*
offline :→ *PeerL*
id : *Peer* →? *PeerId*
addr : *Peer* →? *Address*
preds *isOnline* : *Peer*

where we have (static) sorts, describing data types, like for instance the (totally unspecified) sort for peer identifiers, *PeerId*, or that for the labels of their transitions, *PeerL*. But, we also have the dynamic sort *Peer*, representing the states of

[3] Since the burden of communication in such applications is usually small, the overlay network is often built on a simpler client-server architecture. Nevertheless, the presence of a server is made transparent both to the user and at a given abstraction level in the application.
[4] Notice the obvious adaptation to the case with silent information.

the nodes. Analogously, we have operations building some sort, like for instance *online* and *offline*, which denote particular labels, and we have observers, both operations like *id* and *addr*, and predicates like *isOnline*, used to extract, or observe, aspects of the peer states.

Now, we need to state two different kinds of axioms. First of all, we have the standard axioms, describing the effects of operations and transitions, such as asserting that after going online with an address a, the peer is actually online and its address is a. But, we also have to state that no transition is affecting the value of *id*, as the identity is persistent, that the only transitions affecting *isOnline* are those actually taking the peer on and off line, and that the address is persistent for each connection, so that it can change only if some connection or disconnection has taken place. In other words, we have to state a sort of *frame assumption* for some observers[5]. These are quite different from the previous ones, from a logical point of view, because they express a property that the users usually implicitly assume: *each aspects of the status of the system changes only if forced to, by a transition explicitly modifying it.* But, there is no such a thing as an *implicit* assumption in specifications. Unless some axiom is imposed to guarantee it, there are models which do not satisfy it. Moreover, from a technical viewpoint, they require the end user to add lots of trivial axioms of the form $i : d \xrightarrow{l} d' \Rightarrow p(d) = p(d')$ to state that the transition (s)he is introducing does not affect the result of most observers. This is mostly inconvenient, because usually a very restricted number of transitions may affect the result of an observer and, methodologically, the user is more encouraged to focus on the pairs "transition + observer" where the transition is relevant to that observer than on those where, being no relationship between the two components, things are not going to change and hence the corresponding axiom has to be issued.

In the following section, we will introduce some syntactic sugar intended to help with this issue, which presented itself in most specifications in our library.

2.1 A Spoonful of Sugar

The most natural description of what we want to specify would involve higher-order logic. Indeed, it suffices to decide which operations on dynamic sorts are observers, by a predicate *on the operations*, and axiomatize the capability of the transitions, represented by their information and label components, of affecting the observer result. Such capability would be naturally described by a predicate *on observers,* label and information sorts. Unfortunately, the higher-order features and the dynamic features are added to CASL by two distict extensions: HOCASL and CASL-LTL. Thus, we cannot have both (without defining a super-extension of both languages). In order to avoid the need for second-order logic, we have implemented an analogous mechanism at the first-order level, by adding

[5] In our approach, we do not require a full fledged frame assumption. Indeed, we want to explicitly state that some properties of the system change and some do not, but leave most of them underspecified, changing or not depending on the individual models.

a predicate for each observer, representing the capability of transitions of affecting that aspect. These predicates are required to be freely constructed over a set of axioms, so that their minimal truth is guaranteed. For instance in the case of the *peer* specification, we will have an *_aff_isOnline* predicate and axioms stating that the labels *online* and *offline* do affect it. Then, the predicate is required to be free, so that it is false on all other labels.

However, this approach, would prevent us to introduce later on other labels affecting the predicate. Following our observational approach, instead of stating that some individual label is affecting a predicate, we describe abstract properties on the labels such that the labels satisfying them are those which could affect the predicate. For instance, in our example, the property of being online may be influenced by all the labels representing a connection or a disconnection, but by no others. Thus, we use again predicates on the labels and info to describe the category of action they are representing and use these predicates in turn to state the axioms for the definition of the affecting predicates; then the actual labels become usually superfluous and can be dropped.

This mechanism allows to clearly separate the axioms stating which category of transitions affects which aspects from those describing the effects; moreover, the axiomatization of the default behaviour, where the observer values are not changing unless some transition affecting the corresponding aspect takes place, may be automatically added.

Therefore, let us introduce a syntactic short-cut, which does not require any change in the semantics of CASL, because the terms introduced by this new construct reduces to terms in standard CASL-LTL. In the choice of the restrictions for such a construct, we have been guided by pragmatic considerations, choosing a generality sufficient to deal with all the cases in our library and, at the same time, not so extreme to make the translation in standard CASL difficult.

Let us introduce the notion of *observer block*. The idea is to collect together the definition of observers and the decisions about which category of transition may affect the observer result. Then, by requiring the freeness of the predicate for observer modifiers, we automatically get that all the transition labels and information not explicitely listed as possibly affecting an observer are not allowed to affect it.

Definition 1. *Given the declaration* ***dsort*** *ds* ***label*** *l_ds* ***info*** *i_ds of a dynamic sort, an* observer block *for ds is bracketed between the keywords* ***obs*** *and* ***end_obs****, and it consists of three parts:*

- *a declaration of operations and or predicates, having ds as (unique) source, called* observers on *ds;*
- *a declaration of predicates, having $l\_ds \times i\_ds$ as source, called* categories on $l\_ds \times i\_ds$*, prefixed by the keyword* ***cats****;*
- *a list of axioms (and variable declarations), of the form* $\mathrm{p}(l, i) \Rightarrow (l, i)$ affects o*, or of the form* (l, i) affects $o \Rightarrow (l, i)$ affects o'*, where*
  - *o and o' are observers on ds, declared in the current block;*
  - *i is a variable of sort i_ds, and analogously l is a variable of sort l_ds;*
  - p *is a category on $l\_ds \times i\_ds$, declared in the current block.*

In each basic specification at most one observer block may appear.

Let us consider as an example the peer specification, using the syntactic sugar introduced so far to represent the observers. Notice that the operations *online* and *offline* have been dropped, because their role is filled by the corresponding predicates. Moreover, we give here the full specification, with also the axioms external to the block. Finally, note that the specification is parametric over the definition of the addresses (e.g., IPv4, IPv6, JXTA or Chord identifiers, etc.)

spec PEER[**sort** *Address*]=
sort *PeerId*
dsort *Peer* **label** *PeerL*
preds *isInitial*: *Peer*
obs
  ops *id* : *Peer* →? *PeerId*
    addr : *Peer* →? *Address*
  preds *isOnline* : *Peer*
  cats *goesOnline* : *PeerL*
    goesOffline : *PeerL* *l* : *PeerL*
  axioms ∀ *l* : *PeerL*
  - $goesOnline(l) \Rightarrow l$ *affects* $isOnline$
  - $goesOffline(l) \Rightarrow l$ *affects* $isOnline$
  - l *affects* $isOnline \Rightarrow l$ *affects* $addr$

end_obs
axioms $\forall\, l : PeerL; \forall\, p, p' : Peer$
- $\neg isInitial(p')\, if\, p \xrightarrow{l} p'$
- $\neg isOnline(p)\, if\, isInitial(p)$
- $isOnline(p')\, if\, p \xrightarrow{l} p' \wedge goesOnline(l)$
- $\neg isOnline(p')\, if\, p \xrightarrow{l} p' \wedge goesOffline(l)$
- $def(addr(p))\, if\, isOnline(p)$

end
Now, let us define the semantics of our constructs, by reduction to CASL-LTL

Definition 2. *A correct*[6] *observer block*
obs
  ops $f_1 : ds \to ?\ s_1; \ldots f_n : ds \to ?\ s_n;$
  preds $p_1, \ldots, p_m : ds;$
  cats $pt_1, \ldots, pt_k : l\_ds \times i\_ds$
  axioms $\varphi_1 \ldots \varphi_h$
end_obs
expands to
ops $f_1 : ds \to ?\ s_1; \ldots f_n : ds \to ?\ s_n;$
preds $p_1, \ldots, p_m : ds; pt_1, \ldots, pt_k : l\_ds \times i\_ds$
  $\_aff\_f_1, \ldots, \_aff\_f_n, \_aff\_p_1, \ldots, \_aff\_p_m : l\_ds \times i\_ds$
axioms %% transitions not affecting $f_1 \ldots f_n, p_1 \ldots p_n$ leave the observer result unchanged
  $(\neg\_aff\_f_1(l, i)) \wedge i : d \xrightarrow{l} d' \Rightarrow f_1(d) = f_1(d')$

[6] We are using only partial functions for simplicity, but total functions are allowed as well, of course.

...
$(\neg\_aff\_p_m(l,i)) \wedge i : d \xrightarrow{l} d' \Rightarrow (p_m(d) \Leftrightarrow p_m(d'))$

Moreover, at the end of the largest basic spec enclosing the block, the following fragment is added, where trans *transforms each occurence of* (l, i) affects o *into* _aff_$o(l, i)$*:*

and {
sorts l_ds,i_ds
preds $pt_1, \ldots, pt_k : l\_ds \times i\_ds$
then free {**preds** $\_aff\_f_1, \ldots, \_aff\_f_n, \_aff\_p_1, \ldots, \_aff\_p_m : l\_ds \times i\_ds$
axioms $trans(\varphi_1) \ldots trans(\varphi_h)$}}

It is worth pointing out that some inconsistency may arise if at the same time

- the same sort for label and info is used for different dynamic sorts and
- observers by the same name are defined for two or more of such dynamic sorts.

But, in our experience we never encountered such a case. Thus, we prefer to keep the syntactic sugar simple, even if it is not working for general (but uncommon) cases.

Let us see what is the expansion of our running example.

spec PEER[**sort** *Address*]=
sort *PeerId*
dsort *Peer* **label** *PeerL*
ops $id : Peer \to ?\ PeerId$
$addr : Peer \to ?\ Address$
preds $isOnline, isInitial : Peer$
$goesOnline, goesOffline : PeerL$
$\_aff\_id, \_aff\_addr, \_aff\_isOnline : l\_ds \times i\_ds$
axioms $\forall\ l : PeerL;\ \forall\ p, p' : Peer$

- $\neg\_aff\_id(l,i) \wedge p \xrightarrow{l} p' \Rightarrow id(p) = id(p')$
- $\neg\_aff\_addr(l,i) \wedge p \xrightarrow{l} p' \Rightarrow addr(p) = addr(p')$
- $\neg\_aff\_isOnline(l,i) \wedge p \xrightarrow{l} p' \Rightarrow (isOnline(p) \Leftrightarrow isOnline(p'))$

- $\neg isInitial(p')\ if\ p \xrightarrow{l} p'$
- $\neg isOnline(p)\ if\ isInitial(p)$
- $isOnline(p')\ if\ p \xrightarrow{l} p' \wedge goesOnline(l)$
- $\neg isOnline(p')\ if\ p \xrightarrow{l} p' \wedge goesOffline(l)$
- $def(addr(p))\ if\ isOnline(p)$

end
and {
sorts *PeerL*
preds $goesOnline, goesOffline : PeerL$
then free { **preds** $\_aff\_id, \_aff\_addr, \_aff\_isOnline : PeerL$
axioms $\forall\ l : PeerL;\ \forall\ p, p' : Peer$

- $goesOnline(l) \Rightarrow \_aff\_isOnline(l)$
- $goesOffline(l) \Rightarrow \_aff\_isOnline(l)$
- $\_aff\_isOnline(l) \Rightarrow \_aff\_addr(l)$ } }

Note that atoms of the form (l, i) *affects* o are well-formed only inside the observer block where o is introduced. Thus, the information about which category of actions may change the value of an observer must be collected all together in that block. In particular, it is useless to redeclare the same observer in a different block, because the corresponding *affect* predicate is already completely defined by the *free* statement; thus, any further axiom cannot change it.

On the contrary, it is possible to change the definition of the category predicates. Thus, labels and info introduced further on in the specification can affect an observer already defined.

3 A Hierarchy of Specifications for Distributed Systems

We have developed a set of specifications with the goal of reflecting the essential facilities of most deployed P2P applications. In this section we will have a glance at the work, explaining our design choices and giving some examples.

3.1 Goal

The purpose of this work is to create an infrastructure that can be used to describe the characteristics of middleware software used for constructing P2P apps. Features of existing peer-to-peer middleware vary broadly. Thus, we have tried to create a structure that can be successfully used to represent characteristics of a broad majority among them.

Mirroring middleware, we aim at creating specifications that can be used at an *intermediate level*. On one hand, our specifications build on lower-level ones. Indeed, we use standard CASL libraries for things such as basic and structured datatypes, and we assume a specification for basic networking aspects, such as addresses or messages. On the other hand, we expect specifications of applications to be built using (part of) our infrastructure.

3.2 Design Guidelines

Generality: our specifications reflect a common base of many different architectures. We expect that real-world application specifications will be more detailed and will fit in as specializations of our abstract ones.

Modular structure: to reflect the fact that P2P applications have very different requirements and implementations, we have broken down functionalities in different specifications that depend on each other, similarly to what happens in software libraries.

Loose specifications: we want our specifications to be useful in the broadest possible field, so our goal is not to give strict specifications that will rule out all implementations that don't satisfy some goal (that will be the library user's duty). Our ultimate goal is that any distributed system should easily be modeled using our infrastructure, providing an effective "shortcut" to the library implementor. Thus, we give high-level specifications that can be specialized to reflect real-world applications.

Incremental philosophy: P2P middleware varies heavily. Many applications have very low requirements (for instance, message reception by the recipient may not be guaranteed). For this reason, we start with simple specifications with little, if any, guarantees, and extend them for the cases in which these guarantees are needed.

Problem-driven approach: to guide ourselves in designing an infrastructure that can be useful for real applications, we have constructed the specification in a problem-driven way: we have chosen a set of application domains[7], seeking to reflect some of the areas in which P2P applications can be useful, and we have designed the specifications to meet them[8].

We think that the good amount of reuse we obtained in the specification proves that these desires are substantially met: very different kind of applications (such as instant messaging and distributed file systems), thanks to the abstract nature of the specification, share much of the specification infrastructure.

3.3 Library Modules

Our resulting specification lies at a very high-level, where many things - such as the nature of the underlying network, or the time needed for delivery of messages to offline nodes - are left unspecified. Moreover, we have not specified anything about the nature of messages or addresses in the network: they have dotted borders in the graph, meaning that we have not specified any characteristics about them in our library.

We have chosen how to divide functionality, using specification as building blocks, trying to be as general as possible and dividing functionalities into small parts. This way, we have isolated some components that can be (and are) reused in describing different applications.

We make heavy use of parameters. This way, the user of the library can instantiate a specification which has a loose formal parameter by using a stricter actual parameter. This mechanism can be used to easily require additional features or particular behaviour restrictions from the software. Let us, for instance, see how we can use PEER as a parameter to build the NET specification.

The NET Specification NET (figure 1), alongside with PEER, is a basic building block for our specifications. It is meant to describe behaviour from a global point of view, whereas in PEER we see what happens on a single node.

A new dynamic sort, *Net*, is specified. Its transitions have no label, since labels specify the interactions of a dynamic system with the outside world, and

[7] The applications chosen are two different kinds of file-sharing applications, one using a Gnutella-style broadcasting search, and the other one using a distributed hash table, an instant messaging application, and a distributed file system.

[8] They are still abstract, in the sense that most of the implementation details (e.g., the particular hashing function used or the scheduling politics) and the user interface are still left unspecified.

spec NET[BASEPEER] =SET[**sort** *Peer*], MAP[**sort** *PeerId*][**sort** *PeerL*] **then**
%% A net can be seen as a set of peers; they join the net via the
%% *online* transition and disconnect via *offline*.
%% A net's transition is the sum of all the transitions done by its
%% peers.
%% We use **info** since a net does not give any information to the
%% outside of the net.

sort $NetI = Map[PeerId][PeerL]$
dsort *Net* **info** *NetI*
preds $isInitial : Net$
$\_\_ \xrightarrow{\_\_} \_\_ \; in \; \_\_ : \_\_ \to \_\_ : Peer \times PeerL \times Peer \times NetI \times Net \times Net$
op $peers : Net \to Set[Peer]$

axioms $\forall \; a : Address; i_p : PeerId; l, l_2 : PeerL; p, p', p_2 : Peer; n, n' : Net; i : NetI$
- $\neg isInitial(n) \; if \; i : n \to n'$

%% This predicate is used only as a shortcut.
- $p \xrightarrow{l} p' \; in \; i : n \to n' \Leftrightarrow$
$i : n \to n' \wedge p \; \epsilon \; peers(n) \wedge [id(p)/l] \; \epsilon \; i \wedge p' \; \epsilon \; peers(n') \wedge p \xrightarrow{l} p'$

%% *peers* gives the set of peers currently connected to the net;
%% identifiers and addresses are unique.
- $peers(n) = \{\} \; if \; isInitial(n)$
- $p \; \epsilon \; peers(n) \Rightarrow isOnline(p)$
- $p \; \epsilon \; peers(n) \wedge p' \; \epsilon \; peers(n) \wedge id(p) = id(p') \Rightarrow p = p'$
- $p \; \epsilon \; peers(n) \wedge p' \; \epsilon \; peers(n) \wedge addr(p) = addr(p') \Rightarrow p = p'$

%% $peers(n)$ evolves conforming to the transitions present in their
%% info part.
%% Peers that "move" evolve according to their actions.
- $\left(\exists p, p' : Peer \bullet p \xrightarrow{l} p' \; in \; i : n \to n' \wedge i_p = id(p)\right)$
$if \; i : n \to n' \wedge [i_p/l] \; \epsilon \; i \wedge \neg goesOnline(l) \wedge \neg goesOffline(l)$

%% Peers that don't do actions remain in the same state.
- $(p \; \epsilon \; peers(n') \Leftrightarrow p \; \epsilon \; peers(n)) \; if \; i : n \to n' \wedge \neg id(p) \; \epsilon \; dom(i)$

%% Peers that connect and disconnect get in and out of *peers*
%% respectively.
- $(\exists p, p' : Peer \bullet id(p) = i_p \wedge p \xrightarrow{l} p' \wedge p' \; \epsilon \; n') \; if \; i[i_p/l] : n \to n' \wedge goesOnline(l)$
- $(\neg \exists p : Peer \bullet id(p) = i_p \wedge p \; \epsilon \; peers(n')) \; if \; i[i_p/l] : n \to n' \wedge goesOffline(l)$

%% We don't allow the empty transition.
- $\neg empty : n \to n'$

%% $sync(n, p, l, p_2, l_2)$ means that – in n – p does l if and
%% only if p_2 simultaneously does l_2, whereas *sync_left* means
%% p doing l implies p_2 doing l_2, but not vice versa.
- $sync(n, p, l, p_2, l_2)) \Leftrightarrow (p \; \epsilon \; n \wedge p_2 \; \epsilon \; n \wedge i : n \to n' \Rightarrow ([id(p)/l] \; \epsilon \; i \Leftrightarrow [id(p_2)/l_2] \; \epsilon \; i))$
- $sync\_left(n, p, l, p_2, l_2)) \Leftrightarrow (p \; \epsilon \; n \wedge p_2 \; \epsilon \; n \wedge i : n \to n' \Rightarrow ([id(p)/l] \; \epsilon \; i \Rightarrow [id(p_2)/l_2] \; \epsilon \; i))$

end

spec BASENET = NET[BASEPEER]

Fig. 1. The NET specification.

things happening in a net have effect only on nodes which are connected to it and considered to be part of it. The info part of a transition is a mapping that relates peers to the transactions they do. A *peers* operation is given, returning the set of connected nodes, alongside with some shortcuts that make it easier to express properties more clearly and concisely.

We didn't use categories and observers with *Net*, since in our view they are useful using simple and atomic transactions, and are less suited to possibly complex and heterogeneous ones like *Net*'s ones.

The BaseNet specification is just a shortcut for Net[BasePeer], which in turn is Net[Peer[**sort** *Addr*]]. Its purpose is to shorten specification declarations, which could get otherwise quite unwieldy.

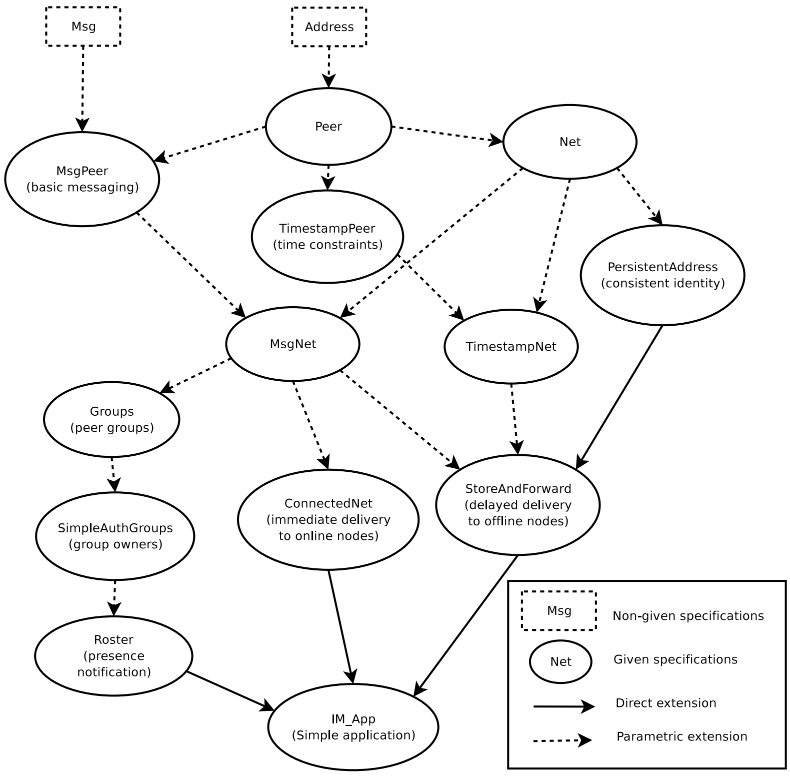

Fig. 2. Hierarchy of specifications for an instant messaging application. Standard Casl libraries are not shown in the graph.

An Example: The Instant Messaging Specification In this section we will have a glance at a self-contained subset of the full library, the part used by the Instant Messaging specification, to see how such a work can be structured.

The rest of the specification library is contained in appendix A. This part of the library can be used to model applications such as ICQ or Jabber [10]. In Figure 2, the dependencies between specifications are shown. At the top of the graph, we have more generic features, that are meant to be used in a wide variety of applications, whereas going towards the bottom, the specification become more and more relevant to the particular application domain we are referring to.

Here is a quick overview of the used specifications.

- We have already seen PEER and NET as the basis of our specifications.
- MSGPEER and MSGNET add to PEER and NET capabilities for messaging. To remain as generic as possible, there are very little constraints here: features such as guaranteed delivery of messages can (and, in this case, will) be required in subsequent specifications. Right now, messages may get immediately received, remain pending for some time, or get lost.

spec MSG[**sort** *Address*] =
%% Messages have an address that is used to recognize the recipient.
sort Msg
op $to : Msg \to Address$
end

spec BASEMSG = MSG[**sort** *Address*]

spec MSGPEER[BASEPEER][BASEMSG] = SET[**sort** *Msg*] **then**
%% We enrich peers with the capability of sending and receiving messages.
%% Thus, labels may carry sets of messages sent/received during the transition.

ops $sent, recvd : PeerL \to Set[Msg]$
axioms $\forall\ l : PeerL; m : Msg; p, p' : Peer$

- $isOnline(p)\ if\ p \xrightarrow{l} p' \wedge isNonEmpty(recvd(l))$
- $to(m) = addr(p)\ if\ p \xrightarrow{l} p' \wedge m \in recvd(l)$

end

spec BASEMSGPEER = MSGPEER[BASEPEER][BASEMSG]

spec MSGNET[BASEMSGPEER][BASENET] =
%% Messages may be immediately received, get lost or remain pending.
%% Peers will only receive messages that have been sent, and not more
%% than once.

op $pending : Net \to Set[Msg]$
axioms $\forall\ i : NetI; l : PeerL; m : Msg; n, n' : Net; p, p' : Peer$

- $pending(n) = \{\}\ if\ isInitial(n)$
- $m \in pending(n) \vee (\exists l : PeerL \bullet l \in range(i) \wedge m \in sent(l))$
  $if\ i : n \to n' \wedge m \in pending(n')$
- $m \in pending(n) \vee (\exists l_2 : PeerL \bullet l_2 \in range(i) \wedge m \in sent(l))$
  $if\ p \xrightarrow{l} p'\ in\ i : n \to n' \wedge m \in recvd(l)$
- $\neg m \in pending(n')\ if\ p \xrightarrow{l} p'\ in\ i : n \to n' \wedge m \in recvd(l)$

end

spec BaseMsgNet = MsgNet[BaseMsgPeer][BaseNet]

– TimestampPeer and TimestampNet let us add time constraint to our specification, adding an underspecified *Stamp* sort, and a *timestamp* observer to peer and net states.

spec TimestampPeer[BasePeer] =
StrictTotalOrder **with sort** *Timestamp* $\mapsto$ *Elem* **then**
%% Useful to indicate time constraints in our specifications.

sort Timestamp
op $stamp : Peer \to Timestamp$
axioms $\forall\ p, p' : Peer; l : PeerL$
- $stamp(p) < stamp(p')\ if\ p \xrightarrow{l} p'$

end

spec BaseTimestampPeer = TimestampPeer[Peer]

spec TimestampNet[BaseTimestampPeer][BaseNet] =
%% The timestamp of a network is defined to be the one of its last-moved
%% node, also counting for peers that have disconnected.

op $stamp : Net \to Timestamp$
axioms $\forall\ p : Peer; i : NetI; l : PeerL; n, n' : Net$
- $stamp(n') = stamp(p')\ if\ p \xrightarrow{l} p'\ in\ i : n \to n'$
- $stamp(n') = stamp(p')\ if\ i : n \to n' \wedge p' \in peers(n') \wedge [id(p')/l] \in i \wedge goesOnline(l)$
- $stamp(n') = stamp(p')\ if\ i : n \to n' \wedge p \in peers(n) \wedge p \xrightarrow{l} p' \wedge [id(p)/l] \in i \wedge goesOffline(l)$

end

spec BaseTimestampNet = TimestampNet[BaseTimestampPeer][BaseNet]

– The PersistentAddress specification states that nodes always use the same address to connect to the net, and no two peers have the same address. This allows other nodes to use addresses of others to identify them, as it happens with email. This is obviously useful for our Instant Messaging application.

spec PersistentAddress[BaseNet] =
%% Peers never change address, and no two different peers may be
%% connected, not even in different times, to the net.
%% This allows other peers to treat address as identifiers.

axioms $\forall\ l : PeerL; n : Net; p, p' : Peer$
- $addr(p') = addr(p)\ if\ p \xrightarrow{l} p'$
- $in\_any\_case(n, [n' \bullet p' \in peers(n') \wedge addr(p) = addr(p') \Rightarrow id(p) = id(p')])$
  $if\ p \in peers(n)$

end

spec BasePersistentAddress = PersistentAddress[BasePeer]

– In ConnectedNet it is requested that messages to online peers are immediately received; there is no intermediate communication with other peers and no message loss if both sender and recipient are online.

spec ConnectedNet[BaseMsgNet] =
%% Peers are directly connected, and if a node is online it immediately
%% receives messages directed to it.

axioms $\forall\ i : NetI; l_1, l_2 : PeerL; m : Msg; n, n' : Net; p_1, p_1', p_2, p_2' : Peer;$

- $m \in recvd(l_2)$
  $if\ p_1 \xrightarrow{l_1} p_1'\ in\ i : n \rightarrow n' \wedge p_2 \xrightarrow{l_2} p_2'\ in\ i : n \rightarrow n'$
  $\wedge\ m \in sent(l_1) \wedge to(m) = addr(p_2)$

end

spec BaseConnectedNet = ConnectedNet[BaseMsgNet]

– StoreAndForward states that undelivered messages never get lost and remain stored somewhere in the net, and uses timestamps to assure that in a given amount of time a peer that stays online will receive them. Since also ConnectedNet will be included by IM_App, in the latter this is going to apply only to messages to offline nodes.

spec StoreAndForward[BaseMsgNet][BaseTimestampNet] =
PersistentAddress[BaseMsgNet]
then
%% A message to a peer which is pending in a moment stamped as ts
%% is guaranteed to be sent before $incr(ts)$ if the peer stays online.

op $incr : Timestamp \rightarrow Timestamp$

axioms $\forall\ l : PeerL; m : Msg; n, n' : Net; i : NetI; p, p' : Peer; ts : Timestamp$

- $ts < incr(ts)$

%% Messages do not disappear

- $p \xrightarrow{l} p'\ in\ i : n \rightarrow n' \wedge m \in sent(l) \Rightarrow$
  $(\exists l_2 : PeerL; p_2, p_2' : Peer$
  $\bullet\ p_2 \xrightarrow{l_2} p_2'\ in\ i : n \rightarrow n' \wedge addr(p_2) = to(m) \wedge m \in recvd(l_2))$
  $\vee\ m \in pending(n')$
- $i : n \rightarrow n' \wedge m \in pending(n) \wedge p \in peers(n) \wedge addr(p) = to(m) \Rightarrow$
  $m \in recvd(lookup(id(p), i)) \vee m \in pending(n')$

%% If a message is pending at time ts, we get to a state where it came
%% to destination or the addressee is offline before $incr(ts)$

- $in\_any\_case(n, eventually\ [n' \bullet stamp(n') < incr(stamp(n))$
  $\wedge\ ((\neg\exists p' : Peer \bullet p' \in peers(n') \wedge addr(p') = to(m)) \vee \neg m \in pending(n'))])$
  $if\ m \in pending(n)$

end

spec StoreAndForward =
BaseStoreAndForward[BaseMsgNet][BaseTimestampNet]

– Groups adds the notion of *peer groups*: members of a given one group are specialized in some way, dependent of the nature of the used application. SimpleAuthGroups adds an owner to each group, and any peer wishing to join a group has to be authorized by its owner. The Roster[9] specification uses groups to model the set of the "friend" peers for each node that are notified when they change their online/offline status.

spec Groups[BaseMsgNet] **given** Set[**sort** *GroupId*] =
%% Peers may join and leave groups. Each peer group has an address
%% of the same kind of peer addresses; whenever a message is sent to a
%% group, it gets forwarded to its connected members.

ops $joined, left : PeerL \to Set[GroupId]$
$addr : GroupId \to Address$
$forward : Msg \times Address \to Msg$
obs
op $groups : Peer \to Set[GroupId]$;
cats $joins, leaves : PeerL$;
$\forall\ l : PeerL$
• $joins(l) \Rightarrow l$ *affects groups*
• $leaves(l) \Rightarrow l$ *affects groups*
end_obs

axioms $\forall\ a : Address; g : GroupId; i : NetI; l : PeerL; m : Msg; n, n' : Net; p, p', p_2 : Peer$
• $groups(p') = groups(p) - left(l) + joined(l)\ if\ p \xrightarrow{l} p'$
• $left(l) = \{\}\ if\ \neg leaves(l)$
• $joined(l) = \{\}\ if\ \neg joins(l)$
• $to(forward(m, a)) = a$
• $forward(m, addr(p_2))\ \epsilon\ sent(l)$
$if\ p \xrightarrow{l} p'\ in\ i : n \to n' \wedge m\ \epsilon\ sent(l) \wedge addr(g) = to(m)$
$\wedge\ p_2\ \epsilon\ peers(n) \wedge g\ \epsilon\ groups(p_2)$
end

spec BaseGroups = Groups[BaseNetMsg]

spec SimpleAuthGroups[BaseGroups] =
%% Very simple specification for group authorization.
%% Groups have an owner, and peers need a sinchronized permission from
%% its owner to join a group.

op $owner : GroupId \to PeerId$
pred $permits : PeerL \times GroupId \times PeerId$

axioms $\forall\ g : GroupId; i : NetI; l : PeerL; n, n' : Net; p, p' : Peer$
• $permits(lookup(owner(g), i), g, id(p))\ if\ p \xrightarrow{l} p'\ in\ i : n \to n' \wedge g\ \epsilon\ joined(l)$
end

[9] *roster* is jargon for contact lists that receive notifications of a node's presence.

spec BaseSimpleAuthGroups = SimpleAuthGroups[BaseGroups]

spec Roster[BaseSimpleAuthGroups] = String **then**
%% Instant messaging basically involves sending simple messages and
%% presence notification to peers in one's roster. In this case, we use
%% online and offline notification, and simple text messages.

ops $roster : PeerId \to GroupId$
$online, offline : PeerId \to Msg$
$plain : String \times Addr \to Msg$

axioms $\forall\ a : Address; i_p : PeerId; i : NetI; n, n' : Net; s : String$
- $owner(roster(i_p)) = i_p$
- $to(plain(s, a)) = a$
- $to(online(i_p)) = addr(roster(i_p))$
- $to(offline(i_p)) = addr(roster(i_p))$
- $online(i_p) \in sent(l)\ if\ p \xrightarrow{l} p' \wedge goesOnline(l)$
- $offline(i_p) \in sent(l)\ if\ p \xrightarrow{l} p' \wedge goesOffline(l)$

end

spec BaseRoster = Roster[BaseSimpleAuthGroups]

– The final specification uses Casl's extension mechanism to combine the previous specification in a neat way, with no need for additional "glue".

spec IM_App =
BaseRoster **and** BaseStoreAndForward **and** BaseConnectedNet
%% Simple IM application.
reveal $sent, recvd, plain, roster, joined, left,$
$online, offline, goesOnline, goesOffline$
end

The strategy for giving a gradually more detailed specification is to extend our more generic specifications with others that are stricter. It is to be noted that our building blocks are already reusable: for instance, the PersistentAddress or StoreAndForward specifications aren't included in our file-sharing specifications, but if they were, they would have reflected a more reliable kind of applications.

3.4 Using Libraries in the Real World

The libraries we have given can be refined to an arbitrary extent, and using parameters the refined specifications can easily be integrated in our already existing framework. In order to have a complete specification, we will have to provide more detailed specifications, restricting our building blocks to actual implementations done in real middleware. These will be used by the user in a later phase of the design, when the platform choice has been done.

For instance, let us see a specification modeling a simple dial-up peer.

from BASIC/STRUCTUREDDATATYPES **get** STRING

spec 32BITINT =
sort *32BitInt*; %% We suppose to have the trivial definition here.
end

spec DIALUP = 32BITINT, STRING, PEER[**sort** *32BitInt*] **then**
ops $call : String \to PeerL$
 $hangup, online :\to PeerL$
 $providerPhone :\to String$
obs
 pred $usingLine : Peer;$
 cats $takesLine, releasesLine : PeerL;$
 $\forall\ l : PeerL$
 • $takesLine(l) \Rightarrow l\ affects\ usingLine$
 • $releasesLine(l) \Rightarrow l\ affects\ usingLine$
end_obs
axioms $\forall\ l : PeerL; p, p' : Peer; s : String$
%% Define which labels pertain to which categories.
 • $goesOnline(online)$
 • $goesOffline(hangup)$
 • $takesLine(call(s))$
 • $releasesLine(hangup)$
 • $usingLine(p')\ if\ p \xrightarrow{l} p' \wedge takesLine(l)$
 • $\neg usingLine(p')\ if\ p \xrightarrow{l} p' \wedge releasesLine(l)$
%% A peer can only call the provider's number.
 • $s = providerPhone\ if\ p \xrightarrow{call(s)} p'$
%% A peer can only connect when it's using the phone.
 • $usingLine(p)\ if\ p \xrightarrow{online} p'$
end

The *Address* sort parameter is instantiated as an IP address, which is in turn a 32-bit integer. A node goes online or offline by calling a telephone number of an internet provider and negotiating an address. We introduce a new observer, *usingLine*, that monitors whether the node is using the telephone line.

Conclusions and Further Work

We have presented a first attempt to a specification library for the representation of middleware for distributed system. We believe that our infrastructure can be useful to model real applications. Indeed, on one hand, it is made of sufficiently generic building blocks to be used in a wide field. and on the other hand, it's easy to extend specifications giving constraints allowing us to reflect more closely a given middleware. Obviously, some specifications given in our infrastructure reflect features that are present in some software and not in others. The designer using our library will be guided by his/her choice of specifications to understand the features needed by the application, and hence helped to chose the appropriate middleware to realize it. We believe that the use of this kind of libraries could help the designer as much as the middleware aids the developer.

Moreover, we have introduced a specification style and some syntactic sugar supporting it in the CASL language, which we think useful when dealing with the definition of loose dynamic specifictions.

In order to make our library fully accessible, we still have to polish and complete it and formally define the CASL variation we are using in it. In particular the tool support (the preprocessor translating our specification in standard CASL) is missing.

Acknowledgements

We thank Gianna Reggio for spending her valuable time in stimulating discussions on the subject of specification of dynamic system and for her kind support.

References

1. Hubert Baumeister and Alexandre V. Zamulin. State-based extension of CASL. In W. Grieskamp, T. Santen, and B. Stoddart, editors, *Integrated Formal Methods, Second International Conference, IFM 2000, Dagstuhl Castle, Germany, Proceedings*, Lecture Notes in Computer Science 1945, pages 3–24. Springer Verlag, 2000.
2. Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, Animesh Nandi, Antony Rowstron, and Atul Singh. SplitStream: high-bandwidth multicast in cooperative environments. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, volume 37, 5 of *Operating Systems Review*, pages 298–313, New York, October 19–22 2003. ACM Press.
3. M. Cerioli. Basic concepts. In *Algebraic System Specification and Development: Survey and Annotated Bibliography. 2nd edition, 1997*, number 3 in Monographs of the Bremen Institute of Safe Systems, chapter 1. Shaker, 1998. ISBN 3-8265-4067-0.
4. Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hongang. Freenet: A distributed anonymous information storage and retrieval system in designing privacy enhancing technologie. In Hannes Federrath, editor, *Designing Privacy Enhancing Technologies*, volume 2009 of *Lecture Notes in Computer Science*, Berkeley, CA, USA, July 2000. Springer-Verlag, Berlin Germany.
5. CoFI (The Common Framework Initiative). CASL *Reference Manual*. Lecture Notes in Computer Science 2960 (IFIP Series). Springer Verlag, 2004.
6. B. Cohen. Incentives build robustness in bittorrent. In *Proceedings of the Workshop on Economics of Peer-to-Peer Systems*, Berkeley, CA, USA, 2003.
7. Dejan S. Milojicic, Vana Kalogeraki, Rajan Lukose, Kiran Nagaraja, Jim Pruyne, Bruno Richard, and Zhichen Xu Sami Rollins. Peer-to-peer computing. Technical report, HP Laboratories Palo Alto, March 2003.
8. Gianna Reggio, Egidio Astesiano, and Christine Choppy. CASL-LTL: A CASL extension for dynamic reactive systems – version 1.0 – summary. technical report DISI-TR-03-36, Univ. of Genova, 2003.

9. H. Reichel. Behavioural equivalence — a unifying concept for initial and final specification methods. In *Proc. 3rd. Hungarian Comp. Sci. Conference*, pages 27–39, 1981.
10. P. Saint-Andre. Extensible messaging and presence protocol (xmpp): Core. http://www.jabber.org/ietf/draft-ietf-xmpp-core-22.html, January 2004.
11. Lutz Schröder and Till Mossakowski. HasCasl: Towards integrated specification and development of Haskell programs. In H. Kirchner and C. Ringeissen, editors, *Algebraic Methods and Software Technology, 9th International Conference, AMAST 2002, Saint-Gilles-les-Bains, Reunion Island, France, Proceedings*, Lecture Notes in Computer Science 2422, pages 99–116. Springer Verlag, 2002.
12. Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for Internet applications. *IEEE/ACM Transactions on Networking*, 11(1):17–32, February 2003.

A Others Specification

This section contains specifications that have not been described in the paper.

from Basic/RelationsAndOrders **get** StrictTotalOrder
from Basic/StructuredDatatypes **get** List, Map, Set, Pair, String

spec SparsePeer[BasePeer] = Set[**sort** *Address*] **then**
%% Peers have a number of direct connections, observed by *neighbors*
%% to other nodes. Note that connected nodes may abruptly and
%% unilaterally disconnect without the peer noticing; that means
%% *neighbors* is a superset of the effectively connected nodes.
%% On a higher level, a keep-alive protocol may be implemented, having
%% the duty of noticing closed connections on peers.

ops $newNeighbors, oldNeighbors : PeerL \to Set[Address]$
obs
  op $neighbors : Peer \to Set[Address]$;
  cats $connecting, disconnecting : PeerL$;
  $\forall\ l : PeerL$
  • $connects(l) \Rightarrow l(n, e)$ *affects ighbors*
  • $disconnects(l) \Rightarrow l(n, e)$ *affects ighbors*
end_obs

axioms $\forall\ l : PeerL; p, p' : Peer$
  • $neighbors(p) = \{\}\ if\ isInitial(p)$
  • $neighbors(p') = neighbors(p) - oldNeighbors(l) + newNeighbors(l)$
    $if\ p \xrightarrow{l} p'$
  • $newNeighbors(l) = \{\}\ if\ \neg connects(l)$
  • $oldNeighbors(l) = \{\}\ if\ \neg disconnects(l)$
  • $neighbors(p) = \{\}\ if\ \neg isOnline(p)$
end

spec BASESPARSEPEER = SPARSEPEER[BASEPEER]

spec SPARSENET[BASESPARSEPEER][BASEMSGNET] =
%% The *neighbors* predicate for nets tells us which pair of peers
%% are really connected.

pred $neighbors : Peer \times Peer$
axioms $\forall\ p_1, p_2 : Peer$
- $neighbors(p_1, p_2) \Leftrightarrow p_1 \epsilon\ neighbors(p_2) \wedge p_2 \epsilon\ neighbors(p_1)$
- $addr(p_1) \epsilon\ newNeighbors(l_2)$
  $if\ p_1 \xrightarrow{l_1} p_1'\ in\ i : n \to n' \wedge p_2 \xrightarrow{l_2} p_2'\ in\ i : n \to n'$
  $\wedge\ addr(p_2) \epsilon\ newNeighbors(l_1)$
- $m \epsilon\ recvd(l_2)$
  $if\ p_1 \xrightarrow{l_1} p_1'\ in\ i : n \to n' \wedge p_2 \xrightarrow{l_2} p_2'\ in\ i : n \to n'$
  $\wedge\ neighbors(p_1, p_2) \wedge m \epsilon\ sent(l_1) \wedge to(m) = addr(p_2)$

end

spec BASESPARSENET = SPARSENET[SPARSEPEER][MSGNET]

spec QUERIES =
%% This specification is meant to be extended to describe queries,
%% resources identifiers, and the way they match.

sorts Query, ResId
pred $match : Query \times ResId$
end

spec RESOURCES[BASEPEER][QUERIES] =SET[**sort** *ResId*] **then**
%% A peer *publish*ing a resource makes it available to other peers.
%% The *vault* observer indicates which resources have been published.

ops $published, removed : PeerL \to Set[ResId]$
obs
  op $vault : Peer \to Set[ResId]$;
  cats $publishes, removes : PeerL$;
  $\forall\ l : PeerL$
- $publishes(l) \Rightarrow l(v, a)\ affects\ ult$
- $removes(l) \Rightarrow l(v, a)\ affects\ ult$

end_obs

axioms $\forall\ l : PeerL; p, p' : Peer$
- $vault(p) = \{\}\ if\ isInitial(p)$
- $vault(p') = vault(p) - removed(l) + published(l)\ if\ p \xrightarrow{l} p'$
- $published(l) = \{\}\ if\ \neg publishes(l)$
- $removed(l) = \{\}\ if\ \neg removes(l)$

end

spec BASERESOURCES = RESOURCES[BASEPEER][QUERIES]

spec SCHEDULERPOLICY[BASENET] =SET[**sort** TASK]

%% A scheduler policy tells us which transitions introduce new duties to
%% be satisfied by the scheduler, and which transitions satisfy them.
%% The *satisfied* and *generated* operations depend on the
%% state of the peer, because in complex cases the scheduler may opt to
%% do different things based on the state of the node (e.g. dropping
%% duties when it's overloaded).

ops $satisfied, generated : Net \times Peer \times PeerL \rightarrow Set[Task]$
end

spec BASESCHEDULERPOLICY = SCHEDULERPOLICY[BASEPEER]

spec SCHEDULER[BASESCHEDULERPOLICY] **given** SET[**sort** *Task*] =
%% The tasks needed to be satisfied are kept track with the *todo*
%% observer.

obs
ops $tasks, done, todo : Peer \rightarrow Set[Task]$;
cats $satisfies, generates : PeerL$;
$\forall\ l : PeerL$
• $satisfies(l) \Rightarrow l(t, o)\ affects\ do$
• $generates(l) \Rightarrow l(t, o)\ affects\ do$
end_obs

axioms $\forall\ l : PeerL; p, p' : Peer$
• $todo(p) = \{\}\ if\ isInitial(p)$
• $todo(p') = todo(p) - satisfied(n, p, l) + generated(n, p, l)\ if\ p \xrightarrow{l} p'\ in\ i : n \rightarrow n'$
• $satisfied(n, p, l) = \{\}\ if\ \neg satisfies(l)$
• $generated(n, p, l) = \{\}\ if\ \neg generates(l)$
end

spec BASESCHEDULER = SCHEDULER[BASESCHEDULERPOLICY]

spec BROADCASTSEARCH[BASESPARSEPEER][BASEMSGPEER][BASERESOURCES] =
SCHEDULERPOLICY **and** LIST[**sort** *Address* =
%% This specification is meant to match a broadcast method of searching
%% of the same type used in Gnutella.
%% *propagate*, *newTTL* and *reduceTTL* can be tailored to mirror
%% real-world applications.

sorts TTL, Route = List[Address]
preds $propagate : Peer \times Query \times TTL \times Route \times Address$
$propagate : Address \times Set[ResId] \times Route$
$createsQuery : PeerL \times Query$
$receivesResults : PeerL \times Set[ResId]$
ops $newTTL : Peer \times Query \rightarrow TTL$
$reduceTTL : Peer \times Query \times TTL \rightarrow TTL$
$query : Query \times TTL \times Route \times Address \rightarrow Msg$
$query : Query \times TTL \times Route \times Address \rightarrow Task$
$results : Address \times Set[ResId] \times Route \rightarrow ?\ Msg$

$results : Address \times Set[ResId] \times Route \to? Task$
$matching : Peer \times Query \to Set[ResId]$

axioms $\forall\ a, a_2 : Address; l : PeerL; n : Net; p : Peer; q : Query; r : ResId; rr : Set[ResId];$
$rt : Route; ttl : TTL$

%% *matching* are published resources on a peer matching a given query.
- $r \in matching(p, q) \Leftrightarrow match(r, q) \wedge r \in vault(p)$

%% *query* and *result* represent "live" queries; the route they pack
%% is the one they've taken so far and will take back to get to the
%% originating node respectively. The task will be satisfied by sending
%% the corresponding message.
- $def\, results(a, rr, rt) \Leftrightarrow \neg isEmpty(rt)$
- $to(queryMsg(q, ttl, rt, a)) = a$
- $to(resultsMsg(a, rr, a_2 :: rt)) = a_2$
- $query(q, ttl, rt, a) \in satisfied(n, p, l)$ *if* $query(q, ttl, rt, a) \in sent(l)$
- $results(a, rr, rt) \in satisfied(n, p, l)$ *if* $results(a, rr, rt) \in sent(l)$

%% New and propagated queries will be sent to each neighbor *propagate*
%% allows us to.
- $query(q, newTTL(p, q), addr(p) :: [], a) \in generated(n, p, l)$
  if $propagate(p, q, newTTL(p, q), addr(p) : [], a) \wedge a \in neighbors(p)$
  $\wedge\ createsQuery(l, q)$
- $query(q, reduceTTL(p, q, ttl), addr(p) :: rt, a) \in generated(n, p, l)$
  if $propagate(p, q, ttl, rt, a) \wedge a \in neighbors(p) \wedge query(q, ttl, rt) \in recvd(l)$

%% Whenever a node receives a query, it sends back the results it has
%% that match with it.
- $results(addr(p), matching(p, q), rt) \in generated(n, p, l)$
  if $isNonEmpty(matching(p, q)) \wedge propagate(addr(p), matching(p, q), rt)$
  $\wedge\ query(q, ttl, rt) \in recvd(l)$

%% Result sets get passed back doing backwards the same route.
- $results(a, rr, rt) \in generated(n, p, l)$
  if $propagate(a, rr, rt) \wedge results(a, rr, addr(p) :: rt) \in recvd(l)$

%% A result set with an empty return route is addressed to the peer who
%% receives it.
- $receivesResults(l, rr)$ *if* $results(a, rr, a_2) \in recvd(l)$

end

spec BaseBroadcastSearch =
BroadcastSearch[BaseSparsePeer][BaseMsgPeer][BaseResources]
end

spec FileManager[BaseResources][**sort** *File*] =
%% A subset of the resources available in a net are files.
%% Peers modifying the file are not necessarily the ones who are
%% storing it.

pred $isFile : ResId$
$sendsFile : PeerL \times ResId \times File \times Address$
$getsFile : PeerL \times ResId \times File$
$writes : PeerL \times ResId \times File$
ops $requestFile : ResId \times Address \times Address \to Msg$
$file : Net \times ResId \to? File$
$empty : \to File$

axioms $\forall\ a, a_1, a_2 : Address; f, f_1, f_2 : File; i : NetI; i_p, i_{p1}, i_{p2} : PeerId;$
$l, l_1, l_2 : PeerL; n, n' : Net; p, p_1, p_2 : Peer; r : Resource$

- $to(requestFile(r, a_1, a_2) = a_2$

%% When a peer sends a file to another, that one is receiving it.
- $getsFile(l_2, r, f)$
  $if\ p_1 \xrightarrow{l_1} p'_1\ in\ i : n \to n' \wedge p_2 \xrightarrow{l_2} p'_2\ in\ i : n \to n'$
  $\wedge\ sendsFile(l_1, r, f, addr(p_2))$

%% A peer can only request files for itself.
- $a_1 = addr(p)\ if\ p \xrightarrow{l} p' \wedge requestFile(r, a_1, a_2) \epsilon\ sent(l)$

%% A file is on the net when some node is storing it.
- $def\ file(n, r)\ if\ isFile(r) \wedge \exists p : Peer \bullet\ \wedge p \epsilon\ n \wedge r \epsilon\ vault(p)$

%% We only send the correct version of a file.
- $file(n, r) = f\ if\ p \xrightarrow{l} p'\ in\ i : n \to n' \wedge sendsFile(l, r, f, a)$

%% No two nodes may modify the same file in the same time.
- $i[i_{p1}/l_1][i_{p2}/l_2] : n \to n' \wedge writes(l_1, r, f_1) \wedge writes(l_2, r, f_2) \Rightarrow i_{p1} = i_{p2}$

%% A new file is created as empty, and is modified via a transition
%% writing from a node.
- $file(n', r) = empty\ if\ i[i_p/l] : n \to n' \wedge r\ \epsilon\ published(l) \wedge isFile(r)$
- $file(n', r) = f\ if\ i[i_p/l] : n \to n' \wedge writes(l, r, f) \wedge isFile(r)$
- $file(n', r) = file(n, r)\ if\ i : n \to n'$
  $\wedge \neg\exists l : PeerL; f : File \bullet\ writes(l, r, f) \wedge l\ \epsilon\ range(i)$

end

spec BaseFileManager = FileManager[BaseResources][**sort** *File*]

spec BroadcastSearch_App =
%% Simple Gnutella-style file-sharing application.
Scheduler[BroadcastSearch
[BaseSparseNet][BaseSparseNet][BaseFileManager]]
reveal *goesOnline, goesOffline, publishes, removes, receivesResults, sends, requestFile, getsFile*
end

spec DHT[BaseMsgPeer][BaseResources] =

PAIR[**sort** *Address*][**sort** *ResId*] **and** SET[**sort** *Hash*] **and** SET[**sort** *Address*]
and SET[**sort** *Pair*[**sort** *Address*][**sort** *ResId*]] **and** SCHEDULERPOLICY **then**

%% We give the same functionalities as in BROADCASTSEARCH,
%% requiring a DHT implementation this time.

sort ResLoc = Pair[Address][ResId]

pred $createsQuery : PeerL \times Query$
$receivesResults : PeerL \times Set[ResLoc]$

ops $lookup : Net \times Hash \to Set[Address]$
$hash : Query \to Set[Hash]$
$hash : ResId \to Set[Hash]$
$pack : Address \times Query \times Address \to Msg$
$pack : Set[ResLoc] \times Address \to Msg$
$send : Msg \to Task$
$resCache : Peer \to Set[ResLoc]$
$matching : Peer \times Query \to Set[ResLoc]$

axioms $\forall\ a : Address; ar : ResLoc; l : PeerL; m : Msg; n : Net; p : Peer; q : Query; r : ResId;$
$rr : Set[ResLoc]$

%% *matching* are the resources on a peer's cache that match a query.

- $pair(a, r) \ \epsilon\ matching(p, q) \Leftrightarrow ar \ \epsilon\ resCache(p) \wedge match(q, r)$

%% We use *hash* both on queries and resources. We need a non-empty
%% intersection for matching queries and resources.

- $isNonEmpty(hash(q) \cap hash(r))\ if\ match(q, r)$

%% The *send* task is satisfied by sending the appropriate message.

- $send(m) \ \epsilon\ satisfied(n, p, l)\ if\ m \ \epsilon\ sent(l)$

%% Nodes keep a cache containing *ResId*s and their location, mantaining
%% info about the resources they heard of. The policy for deleting old
%% entries is not specified here.

- $rr \subseteq resCache(p')\ if\ p \xrightarrow{l} p' \wedge pack(rr, addr(p)) \ \epsilon\ recvd(l)$

%% A node's cache consists of old entries and just received ones.

- $ar \ \epsilon\ resCache(p) \vee (\exists rr : Set[ResLoc] \bullet pack(rr, addr(p)) \ \epsilon\ recvd(l) \wedge ar \ \epsilon\ rr)$
  $if\ p \xrightarrow{l} p' \wedge ar \ \epsilon\ resCache(p')$

%% *lookup* tells us the nodes which are responsible for an hash.
%% When a resource is created, an advertising for it is sent to all
%% nodes responsible for its hash.

- $send(pack(pair(addr(p), r), a)) \ \epsilon\ generated(n, p, l)$
  $if\ p \xrightarrow{l} p'\ in\ i : n \to n' \wedge r \ \epsilon\ published(l)$
  $\wedge\ \exists h : Hash \bullet h \ \epsilon\ hash(r) \wedge a \ \epsilon\ lookup(n, h)$

%% Queries get sent to responsible nodes.

- $send(pack(addr(p), q, a)) \ \epsilon\ generated(n, p, l)$
  $if\ \exists h : Hash \bullet h \ \epsilon\ hash(q) \wedge a \ \epsilon\ lookup(q) \wedge createsQuery(l, q)$

%% Nodes respond to queries with the relevant results.

- $send(pack(matching(p, q), a)) \ \epsilon\ generated(n, p, l)\ if\ pack(a, q, addr(p)) \ \epsilon\ recvd(l)$

• $receivesResults(l, rr)$ *if* $pack(rr, a) \in recvd(l)$
end

spec BaseDHT = DHT[BaseMsgPeer][BaseResources]

spec DHTFileSharing_App =
Scheduler[DHT[BaseConnectedNet][BaseFileManager]]
%% We expose the same operations as in BroadcastSearchApp,
%% but we require a different implementation.
reveal *goesOnline*, *goesOffline*, *publishes*, *removes*, *createsQuery*,
receivesResults, *sent*, *requestsFile*, *getsFile*
end

spec FairScheduling[BaseScheduler] =
%% Assure that each pending task gets eventually accomplished.

axioms $\forall\ n : Net; p : Peer; t : Task$
• $t \in todo(p) \wedge p \in peers(n) \Rightarrow$
$in\_any\_case(n, (\exists l : PeerL; p^x : Peer$
$\bullet\ [n^x \bullet p^x \in n^x] \wedge \langle i \bullet [id(p)/l] \in i\rangle \wedge t \in satisfied(n^x, p^x, l))$
end

spec BaseFairScheduling = FairScheduling[BaseScheduler]

spec ReliableResources[BaseResources] =
%% When a resources has been declared as *available*, it stays so
%% (i.e. there is a peer that can provide it) until it's made
%% *unavailable*.

pred $available : Net \times ResId$
$makesAvailable, makesUnavailable : PeerL \times ResId$

axioms $\forall\ i : NetI; n, n' : Net; p : Peer; r : ResId$
• $available(n, r) \Rightarrow \exists p : Peer \bullet p \in peers(n) \wedge r \in vault(p)$
• $isInitial(n) \Rightarrow \neg available(n, r)$
• $(available(n', r) \Leftrightarrow$
$(\exists l : PeerL \bullet l \in range(i) \wedge makesAvailable(l, r))$
$\vee\ (available(n, r) \wedge \neg(\exists l : PeerL \bullet l \in range(i) \wedge makesUnavailable(l, r)))$
if $i : n \rightarrow n'$
end

spec BaseReliableResources = ReliableResources[BaseResources]

spec ReliableDHT[BaseReliableResources][BaseDHT][BaseScheduler] =
FairScheduling[BaseScheduler] **then**
%% We assure that available resources can always be retrieved by the
%% DHT.

axioms $\forall\ n : Net; q : Query; r : ResId$

%% For each query matching an available resource, the hashing routes the
%% query to a peer knowing where that resource is.

- $available(n,r) \wedge match(q,r) \Rightarrow$
  $(\exists p_1, p_2 : Peer; h : Hash$
  $\bullet\ p_1 \in peers(n) \wedge p_2 \in peers(n) \wedge h \in hash(q) \wedge addr(p_1) \in lookup(h)$
  $\wedge\ pair(addr(p_2, r)) \in resCache(p) \wedge r \in vault(p_2))$

end

spec BASERELIABLEDHT=
RELIABLEDHT[BASERELIABLERESOURCES][BASEDHT][BASESCHEDULER]

spec AUTHGROUPS[BASEGROUPS] =SET[**sort** *GroupId*]
%% Each group has a set of controllers - i.e. peers that can give other
%% ones the authorization to join that group. The first controller of a
%% group is its creator, and each controller can give control privileges
%% to other peers.
%% Note that this specification *doesn't* extend
%% SIMPLEAUTHGROUPS.

preds $creates : PeerL \times GroupId$
$givesAuth, revokesAuth, givesControl : PeerL \times PeerId \times GroupId$
ops $gotAuth, lostAuth, gotControl : PeerL \rightarrow Set[GroupId]$
obs
ops $controlled, authorized : Peer \rightarrow Set[GroupId]$;
cats $getsAuth, losesAuth, getsControl : PeerL$;
$\forall\ l : PeerL$

- $getsAuth(l) \Rightarrow l(a,u)$ *affects thorized*
- $losesAuth(l) \Rightarrow l(a,u)$ *affects thorized*
- $getsControl(l) \Rightarrow l(c,o)$ *affects ntrolled*

end_obs

axioms $\forall\ g : GroupId; i : NetI; i_p : PeerId; l : PeerL; n, n' : Net; p, p' : Peer$

- $gotAuth(l) = \{\}\ if\ \neg getsAuth(l)$
- $lostAuth(l) = \{\}\ if\ \neg losesAuth(l)$
- $creates(l,g) \Rightarrow getsControl(l)$
- $creates(l,g) \Rightarrow g \in gotControl(l)$
- $gotControl(l) = \{\}\ if\ \neg getsControl(l)$

%% $groups(n)$ keeps track of which groups are present in n.

- $groups(n) = \{\}\ if\ isInitial(n)$
- $\neg g \in groups(n)\ if\ p \xrightarrow{l} p'\ in\ i : n \rightarrow n' \wedge creates(l,g)$
- $(g \in groups(n') \Leftrightarrow g \in groups(n) \vee (\exists l : PeerL \bullet l \in range(i) \wedge creates(l,g)))$
  $if\ i : n \rightarrow n'$

%% $controlled(p)$ stores the ids of groups controlled by p.

- $controlled(p) = \{\}\ if\ isInitial(p)$
- $controlled(p') = controlled(p) + gotControl(l)\ if\ p \xrightarrow{l} p'$

%% Only controlling peers can delegate control of a group

- $g \in controlled(p)\ if\ p \xrightarrow{l} p' \wedge givesControl(l, i_p, g)$

%% If a peer is becoming a controller, someone else is giving that
%% privilege to it.

- $\left(\exists l_2 : PeerL; p_2, p_2' : Peer \bullet p_2 \xrightarrow{l_2} p_2' \ in \ i : n \to n' \land givesControl(l, id(p), g)\right)$
  $if \ p \xrightarrow{l} p' \ in \ i : n \to n' \land g \ \epsilon \ gotControl(l)$

%% $authorized(p)$ are the groups in which p is permitted to log in.

- $authorized(p) = \{\} \ if \ isInitial(p)$
- $authorized(p') = authorized(p) - lostAuth(l) + gotAuth(l) \ if \ p \xrightarrow{l} p'$
- $g \ \epsilon \ controlled(p)$
  $if \ p \xrightarrow{l} p' \ in \ i : n \to n' \land (givesAuth(l, i_p, g) \lor revokesAuth(l, i_p, g))$
- $\left(\exists l_2 : PeerL; p_2, p_2' : Peer \bullet p_2 \xrightarrow{l_2} p_2' \ in \ i : n \to n' \land givesAuth(l, id(p), g)\right)$
  $if \ p \xrightarrow{l} p' \ in \ i : n \to n' \land g \ \epsilon \ gotAuth(l)$
- $\left(\exists l_2 : PeerL; p_2, p_2' : Peer \bullet p_2 \xrightarrow{l_2} p_2' \ in \ i : n \to n' \land revokesAuth(l, id(p), g)\right)$
  $if \ p \xrightarrow{l} p' \ in \ i : n \to n' \land g \ \epsilon \ lostAuth(l)$

- $g \ \epsilon \ authorized(p) \ if \ p \xrightarrow{l} p' \land g \ \epsilon \ joined(l)$

end

spec BASEAUTHGROUPS = AUTHGROUPS[BASEGROUPS]

spec FILEPERMISSIONS[BASEGROUPS][RELIABLERESOURCES[BASEFILEMANAGER]
with ops $makesAvailable \mapsto creates, makesUnavailable \mapsto deletes$]=
SET[**sort** $Perm$] **then**

%% Permissions (Read, Write, and Delete) are given, for each file, to
%% groups. A single node has a privilege if it belongs to a group who
%% has it.

generated type $Perm ::= R | W | D$
sort Perms = Set[Perm]
pred $setsPerms : PeerL \times ResId \times Group \times Perms$
ops $perms : Net \times ResId \times GroupId \to Perms$
$perms : Net \times Peer \times ResId \to Perms$
$owner : Net \times ResId \to ? \ PeerId$

axioms $\forall \ g : GroupId; i : NetI; i_p, i_{p1}, i_{p2} : PeerId; l : PeerL; n, n' : Net; p, p' : Peer;$
$pm : Perm; ps : Perms; r : ResId$

%% Files may only be created when they don't exist; no two peers may
%% create the same file at the same time.

- $\neg available(n, r) \ if \ p \xrightarrow{l} p' \ in \ i : n \to n' \land creates(l, r)$
- $i_{p1} = i_{p2} \ if \ i : n \to n' \land creates(lookup(i_{p1}, i), r) \land creates(lookup(i_{p2}, i), r)$

%% The owner of a file is its creator. He/she is the only one who can
%% change its permissions.

- $def \, owner(n, r) \Leftrightarrow available(n, r) \land isFile(r)$

- $owner(n', r) = id(p)$ *if* $p \xrightarrow{l} p'$ *in* $i : n \to n' \wedge creates(l, g)$
- $owner(n', r) = owner(n, r)$ *if* $i : n \to n' \wedge def\, owner(n', r)$
- $owner(n, r) = i_p$ *if* $p \xrightarrow{l} p'$ *in* $i : n \to n' \wedge setsPerms(l, r, g, ps)$

%% Permissions to a group are given when *setsPerms* is true; they are
%% reset when the file is deleted.

- $perms(n, r, g) = \{\}$ *if* $isInitial(n)$
- $perms(n', r, g) = ps$ *if* $i : n \to n' \wedge setsPerms(lookup(i, owner(n, g)), r, g, ps)$
- $i : n \to n' \wedge \neg(\exists ps : Perms \bullet setsPerms(lookup(i, owner(n, g)), r, g, ps)) \Rightarrow$
  $perms(n', r, g) = \{\}$ *when* $creates(lookup(i, owner(n, g)), r)$
  else $\{\}$ *when* $deletes(lookup(i, owner(n, g)), r)$
  else $perms(n, r, g)$

%% The privileges a single peer has are the union of those granted to
%% the groups he belongs to.

- $pm \in perms(n, p, r) \Leftrightarrow (\exists g : GroupId \bullet g \in groups(p) \wedge pm \in perms(n, r, g))$

%% Axioms relating permissions with their respective actions.

- $R \in perms(n, p, r)$ *if* $p \xrightarrow{l} p'$ *in* $i : n \to n' \wedge getsFile(l, r, f)$
- $W \in perms(n, p, r)$ *if* $p \xrightarrow{l} p'$ *in* $i : n \to n' \wedge writesFile(l, r, f)$
- $D \in perms(n, p, r)$ *if* $p \xrightarrow{l} p'$ *in* $i : n \to n' \wedge deletes(r, f)$

end

spec BaseFilePermissions =
FilePermissions[BaseGroups][ReliableResources[BaseFileManager]]
end

spec FileScheduler[BaseScheduler][BaseFileManager] =
ops $sendFile : ResId \times Address \to File$
axioms $\forall\ a, a_1, a_2 : Address; f : File; l : PeerL; p : PeerId; r : ResId$

- $sendFile(r, a_1) \in generated(n, p, l)$ *if* $requestFile(r, a_1, a_2) \in recvd(l)$
- $sendFile(r, a) \in satisfied(n, p, l)$ *if* $sendsFile(l, r, f, a)$

end

spec BaseFileScheduler = FileScheduler[BaseScheduler][BaseFileManager]

spec DistributedFileSystem_App =
%% Specification for a distributed filesystem implementation.
BasePersistentAddress **and**
FilePermissions[BaseAuthGroups]
[ReliableDHT[BaseFileManager][BaseDHT][BaseFileScheduler]]
reveal *creates*, *deletes*,
sends, *requestsFile*, *getsFile*, *writes*,
createsQuery, *receivesResults*,
perm, *setsPerm*, R, W, D,
$\{\} : Perms, \_\_ + \_\_ : Perms \times Perms \to Perms$
end