Node Coordination in Peer-to-Peer Networks

Luigia Petre[1], Petter Sandvik[1,2], and Kaisa Sere[1]

[1] Department of Information Technologies, Åbo Akademi University
[2] Turku Centre for Computer Science (TUCS)
Turku, Finland

Abstract. Peer-to-peer networks and other many-to-many relations have become popular especially for content transfer. To better understand and trust these types of networks, we need formally derived and verified models for them. Due to the large scale and heterogeneity of these networks, it may be difficult and cumbersome to create and analyse complete models. In this paper, we employ the modularisation approach of the Event-B formalism to model the separation of the functionality of each peer in a peer-to-peer network from the network structure itself, thereby working towards a distributed, formally derived and verified model of a peer-to-peer network. As coordination aspects are fundamental in the network structure, we focus our formalisation effort in this paper especially on these. The resulted approach demonstrates considerable expressivity in modelling coordination aspects in peer-to-peer networks.

1 Introduction

In recent years, there has been a trend of moving away from the traditional client-server model in network software towards peer-to-peer networks and other many-to-many relations. Especially when it comes to large scale content transfer, peer-to-peer applications and protocols such as BitTorrent [8] have become popular [24], and even found their way into electronic appliances such as network routers [6] and television sets [26]. In short, the paradigm switch from client-server communication models to BitTorrent-supporting networks amounts to enabling "clients" that are already downloading e.g., video streams, to also become "servers" for other potential clients that may download the same content. The participation of every peer in content communication provides a tremendous increase in the communication efficiency, in the communication model flexibility, and in the content availability. It is therefore highly beneficial to have a thorough understanding of this communication paradigm, to uncover its potential weaknesses and recognise how to avoid them.

Peer-to-peer networking proposes a mixed coordination model among peers. At first sight, it resembles data-based coordination, such as distributed tuple spaces, in that one peer enumerates in a webpage its downloadable material and another peer starts downloading the material of interest, found via the webpage or via a special server called tracker. However, even during downloading, the second peer also becomes a data provider of that material, solely due to its downloading and without any enumeration of downloadable material in a webpage. This resembles event-based coordination where communication between processes (peers) is enabled by events generated when certain state changes appear. This is also reminiscent of the publisher-subscriber model of coordination. This partial adherence of peer-to-peer networking to several coordination models is currently not singular. In [18], the authors propose the separation of the coarse grained (coordination) control flow into several event handlers that coordinate (via events) the mobile applications. In their turns, the event handlers need to communicate with each other, typically implicitly, via shared data. Coordination and concurrency are studied in the context of Prolog [25] by decoupling logic engines and multithreads for efficiency; cooperative constructs are then illustrated for both Linda [7] blackboards and publish/subscribe models. Realtime coordination in dataflow networks is typically asynchronous, but in [16], coordination patterns are proposed which combine synchrony and asynchrony. All these models simply try to address the ever-increasing complexity of contemporary software-intensive systems from various viewpoints. However, due to combining several aspects of several coordination models, peer-to-peer networking is a rather complicated model to analyse. In this paper we focus on this analysis problem.

In order to gain a thorough understanding of peer-to-peer networking, we develop and analyse models of a peer-to-peer media distribution system. In particular, in this paper we focus on modelling how peers in a such a system could discover and interact with each other, i.e., we model inter-peer relations as the basis of the peer-to-peer coordination model. In swarm-like peer-to-peer systems, where peers interact only when interested in the same content, a peer that is unable to receive incoming connections, for instance when it is behind a firewall, is at a serious disadvantage compared to other peers [10]. Extensions to the original BitTorrent protocol such as peer exchange (PEX) and distributed hash tables (DHT) [17] have been developed to alleviate this problem, and we need a reusable, extendable model of peer discovery and connectivity to be able to model these. Peer-to-peer systems and other distributed architectures have been formally modelled before [15,28,29], but our focus here is on creating a reusable formal model of inter-peer relations using BitTorrent as our model protocol.

Based on the formal modelling of peer-to-peer relations, we make the following contributions:

– We propose a formal model for analysing properties of peer-to-peer relations and networking.
– We distribute this model and the proven properties as a correct development from the initial model.
– We put forward the dual coordination nature of the distributed model (both data-driven and control-oriented) and the further applicability of our employed formal methodology.

We develop our models based on the Event-B formal method [2], which offers excellent tool support in form of the Rodin platform [3,11]. When developing models in Event-B, the primary concept is that of abstraction [2], as *models* are created from abstract specifications and then refined stepwise towards concrete implementations. We prove the correctness of each step of the development using the Rodin platform, which automatically generates *proof obligations*. These are mathematical formulas to prove in order to ensure correctness; the proving can be done automatically or interactively using the Rodin platform tool. The immediate feedback from the provers makes it possible to adapt our model to better suit automatic proving, and this ability to interleave modelling and proving is a big advantage of development in Event-B using the Rodin platform. Event-B is currently extending to also incorporate *modularisation methodology* [13]. This essentially amounts to proposing distributed versions for various models and proving the correctness of the distribution via refinement. Consider the example of a peer connection operation involving two nodes. We can specify this feature in Event-B typically within one module (called *machine*) that has data and operations on the data; we can also model various properties of the module and prove their correctness. However, at the implementation phase, the peer connection operation typically involves two modules, corresponding to the two connecting peers that synchronise with each other, so that each peer adds the required reference to the other. The modularisation methodology allows the transformation of the modelled peer connection operation into a distributed addition of links among the two peers.

We proceed as follows. In Section 2 we describe the Event-B formalism and its modularisation extension. In Section 3 we introduce our inter-peer relation modelling and in Section 4 we present our modular approach to this. We elaborate on our contribution in Section 5. We conclude this paper in Section 6 with discussion of our findings as well as future work.

2 Event-B and Its Modularisation Approach

In this section we overview Event-B and its modularisation approach to the extent needed in this paper.

2.1 Event-B

Event-B [2] is a state-based formal method focused on the stepwise development of correct systems. This formalism is based on Action Systems [5,27] and the B-Method [1]. In Event-B, the development of a model is carried out step by step from an abstract specification to more concrete specifications. The general form of an Event-B model is illustrated in Fig. 1. Models in Event-B consist of *contexts* and *machines*. A context describes the static part of a model, containing sets and constants, together with axioms about these. A machine describes the dynamic part of a model, containing variables, invariants (boolean predicates on the variables), and events, that evaluate (via event *guards*) and modify (via event *actions*) the variables. The guard of an event is an associated boolean predicate on the variables, that determines if the event can execute or not. The action of an event is a parallel composition of either deterministic or non-deterministic assignments. Computation proceeds by a repeated, non-deterministic choice and execution of an enabled event (an event whose guard holds). If none of the events is enabled then the system deadlocks. The relationship *Sees* between a machine and its accompanying context denotes a structuring technique that allows the machine access to the contents of the context.

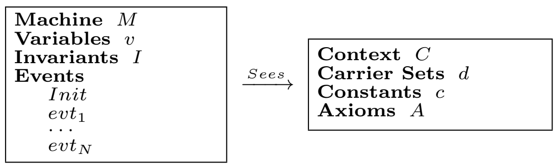

Fig. 1. A machine M and a context C in Event-B

The semantics of Event-B actions is defined using *before-after (BA) predicates* [2,3]. A before-after predicate describes a relationship between the system states before and after the execution of an event. The semantics of a whole Event-B model is formulated as a number of *proof obligations*, expressed in the form of logical sequents. The full list of proof obligations can be found in [2].

System Development. Event-B employs a top-down refinement-based approach to the formal system development. The development starts from an abstract system specification that models some essential functional requirements. While capturing more detailed requirements, each refinement step typically introduces new events and variables into an abstract specification. These new events correspond to stuttering steps that are not visible in the abstract specification. This type of refinement is called *superposition refinement*. Moreover, Event-B formal development supports *data refinement*, allowing us to replace some abstract variables with their concrete counterparts. In that case, the invariant of a refined model formally defines the relationship between the abstract and concrete variables; this type of invariants are called *gluing invariants*.

In order to prove the correctness of each step of the development, a set of proof obligations needs to be discharged. Thus, in each development step we have mathematical proof that our model is correct. The model verification effort and, in particular, the automatic generation and proving of the required proof obligations, are significantly facilitated by the provided tool support – the Rodin platform [3,4].

2.2 The Event-B Modularisation Approach

Recently the Event-B language and tool support have been extended with a possibility to define *modules* [13,12,21] – i.e., components containing groups of callable atomic operations. Modules can have their own external (i.e., global) and internal (i.e., local) state and invariant properties. An important characteristic of modules is that they can be developed separately and, when needed, composed with the main system.

A module description consists of two parts – a *module interface* and a *module body*, the latter being an Event-B machine. Let M be a module. A module interface MI is a separate Event-B component. It allows the user of the module M to invoke its operations and observe the external variables without having to inspect the module implementation details. MI consists of external module variables w, constants c, sets s, the external module invariant $M\_Inv(c, s, w)$, and a collection of module operations O_i, characterised by their pre- and postconditions, as shown in Fig. 2.

```
**Interface** $MI$
 **Sees** $MI\_Context$
 **Variables** $w$
 **Invariants** $M\_Inv(c, s, w)$
 **Initialisation** $\cdots$
 **Process**
     $PE_1 =$ **any** $vl$ **where** $g(c, s, vl, w)$ **then** $S(c, s, vl, w, w')$ **end**
     $\cdots$
 **Operations**
     $O_1 =$ **any** $p$ **pre** $PRE(c, s, vl, w)$ **post** $POST(c, s, vl, w, w')$ **end**
     $\cdots$
```

Fig. 2. Interface Component

In addition, a module interface description may contain a group of standard Event-B events under the **Process** clause. These events model the autonomous module thread of control, expressed in terms of their effect on the external module variables. In other words, the module process describes how the module external variables may change between operation calls.

A formal module development starts with the design of an interface. Once an interface is defined, it is not further developed. This ensures that a module body may be constructed independently from a model relying on the module interface. A module body is an Event-B machine that implements the interface by providing a concrete behaviour for each of the interface operations. A set of additional proof obligations are generated to guarantee that each interface operation has a suitable implementation.

When the module M is imported into another Event-B machine (which is specified by a special clause **USES**), the importing machine can invoke the operations of M and read the external variables of M. To make a module specification generic, in $MI\_Context$ we can define some constants and sets (types) as parameters. The properties over these sets and constants define the constraints to be verified when the module is instantiated. The concrete values or constraints needed for module instantiation are supplied in the **USES** clause of the importing machine.

Module instantiation allows us to create several instances of the same module; we distinguish among these instances using a certain $prefix$. Different instances of a module operate on disjoint state spaces. Via different instantiation of generic parameters the designers can easily accommodate the required variations when developing components with similar functionality. Hence module instantiation provides us with a powerful mechanism for reuse.

The latest developments of the modularisation extension also allow the developer to import a module with a given concrete set as its parameter. This parameter becomes the index set of module instances. In other words, for each value from the given set, the corresponding module instance is created. Since each module instance operates on a disjoint state space, parallel calls to operations of distinct instances are possible in the same event.

3 Modelling Inter-Peer Relations

We illustrate the first three steps of our development of a formal model for inter-peer relations in Fig. 3. In this section we shortly describe this Event-B model in order to facilitate an easier understanding of the modularised model described in the next section. More details can be found in our technical report [20].

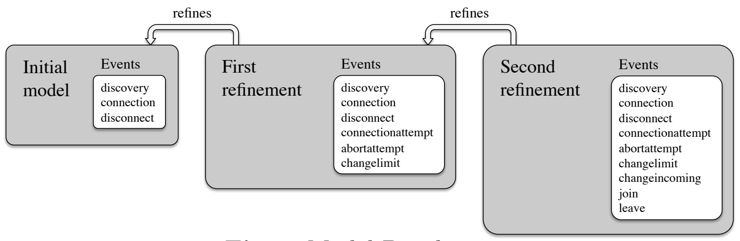

Fig. 3. Model Development

Our initial model is very abstract, with only two major functions. The first concerns one peer becoming aware of other peers. In a peer-to-peer network such as BitTorrent, this would correspond to receiving a list of other peers from a tracker, i.e., a server that keeps track of which peers are involved in sharing a particular content. However, at this stage we are not interested in the specifics of how this subset of all peers is retrieved, only that there is a way of peers to discover other peers. We also note that the tracker is an instantiation of the publish/subscribe coordination model. The second major function is to create a connection between a peer and another peer, where the first peer must be aware of the second but not necessarily vice versa. To model these functions, we define relations between peers, assuming peers are represented by natural numbers for simplicity. An "awareness" relation from 1 to 2 thereby means that peer 1 is aware of peer 2, which is different from a relation from 2 to 1. For the "connection" relation, we note that in practice we only have one connection between two peers, because in peer-to-peer networks such as those based on BitTorrent, connections are symmetrical and traffic can flow in both directions [9]. For that reason, we allow only one connection per peer pair here, e.g., if a "connection" relation exists from 1 to 2 we do not allow one from 2 to 1.

Our initial model is therefore composed of the following events, besides the obligatory *initialisation* event: *discovery*, which creates "awareness" relations from a peer to a subset of other peers, *connection*, which creates a "connection" relation between a peer and another if there is an "awareness" relation from the first to the second, and *disconnect*, which removes an existing "connection" relation between two peers. This disconnection could occur because of network issues or because the peer has decided to no longer participate in the swarm. However, peers also close connections that have had no traffic for a while; Iliofotou et al claim that the differences in download speed between BitTorrent clients can be partly attributed to differences in when they decide to close connections [14]. For this reason it is important for us to model a *disconnect* event that later can be refined into different types of disconnection events. The situation in which peers become unaware of each other does not exist in the actual peer-to-peer networks we are interested in, and therefore there is no need for an event that models such a situation.

For our first refinement step, we limit the amount of connections a peer can have, because otherwise every peer would eventually end up being connected to all the other peers. While this would be possible when the number of peers is low, it would be unrealistic for a large system, and we therefore introduce a connection limit specific to each peer. This means that a connection between two peers may not always be possible, and therefore we also need to modify our connection functionality. Because peers do not know whether another peer can accept their connection or not, we replace our single connection event with two events. The *connectionattempt* event takes a peer whose connection limit has not been reached and another peer that the first peer is aware of but not connected to, and adds a "connection attempt" relation from the first peer to the second one. The *connection* event here takes a peer whose connection limit has not been reached and another peer such that there is a "connection attempt" relation from the second to the first, and creates a "connection" relation from the second to the first while removing the corresponding "connection attempt" relation. We also add another event, *abortattempt*, for aborting a connection attempt, which in practice would happen after a time limit. Because the connection limit is not necessarily constant and can vary between peers, we also add the abstract *changelimit* event describing how the limit may change. The total amount of connections for a peer, specified by the variable *connections*, is here taken to be the sum of the amount of "connection" relations to and from the peer, and the amount of "connection attempt" relations originating from the peer. This means that the limit on connections is a limit on the amount of simultaneous active successful connections and unsuccessful connection attempts.

In the second refinement step we introduce the concept of peers not being able to accept incoming connections, i.e., not being able to have "connection" relations from another peer to itself. First we achieve this in an abstract way, by simply having a boolean variable for each peer and checking the value of that variable before allowing the connection to be created. We add the abstract event *changeincoming* to be able to change the value of this boolean variable for each peer. Later we can refine this situation by specifying a set of more complex relations, such as in the real-life situation where two peers are behind the same firewall and thereby able to accept incoming connections from each other but not from other peers. Furthermore, we refine our model to include *join* and *leave* events for when peers join and leave the swarm, respectively. To reduce the complexity of our model, we specify that all the connections to and from a peer, as well as all connection attempts made by the peer, must be removed before the peer can leave. This can be seen in the following Event-B code:

```
EVENT leave ≙
    any
        peer
    where
        grd1 : $peer \in peers \wedge peer \in onlinepeers$
        grd2 : $\forall p, r \cdot (\{p \mapsto r\} \in connection) \Rightarrow (p \neq peer \wedge r \neq peer)$
        grd3 : $\forall p, r \cdot (\{p \mapsto r\} \in connectionattempt) \Rightarrow (p \neq peer)$
    then
        act1 :  $onlinepeers := onlinepeers \setminus \{peer\}$
    end
```

So far, we have described a monolithic model of inter-peer relations in a peer-to-peer network. Our next step is to use the modularisation approach described in Section 2.2 to separate the internal functionality of a peer from the coordinating functionality of the network structure.

4 Modularising Inter-Peer Relations

Our intent with modularising our model of inter-peer relations is to separate the internal functionality of each peer from the functionality of the network itself; this makes the peers, in a sense, independent of other peers. As we specify the interface that a peer presents to the coordinating network, we can continue to refine and implement the peer separately from the network coordination structure. Therefore, we need to consider which events from our previous model should be implemented in the peer module and which in the Event-B machine specifying the network coordination.

We note that the events *changelimit* and *changeincoming* affect only one peer at a time, and thus should be modelled as processes internal to the peer. Likewise, the *discovery* event only adds to one peer's view, and although it could be argued that this is an event concerning network coordination, nothing specifies that this event needs to invoke the network at all. In BitTorrent, for instance, peer discovery never depends on how peers connect to each other, and therefore it should be seen as a process internal to the peer in this context. The *join* and *leave* events also only affect one peer's status, because we require that the *leave* event is enabled only when the peer has no connections and no connection attempts. This is also reflected in the identically named process in the peer interface, which can be compared to the *leave* event shown in Section 3.

```
PROCESS  $leave \mathrel{\widehat{=}}$
    when
        grd1 :  $isonline = TRUE$
        grd2 :  $connection = \varnothing \land connectionattempt = \varnothing$
    then
        act1 :  $isonline := FALSE$
    end
```

Regarding the *connectionattempt* and *abortattempt* events, we note that these events model the intent of one peer, and thus should be modelled as an event internal to the peer, although the variables modified will be read by the network coordination structure. The remaining events *connection* and *disconnect* require coordination between peers, and thus we will describe in more detail how the equivalent functionality is implemented in the distributed model. The overall structure of this decomposition refinement can be seen in Fig. 4.

As mentioned in Section 2.2, we use the **USES** clause to import an interface into an Event-B machine, specifying the name of the interface, the indexing set, and the prefix used to access varibles and operations from the interface.

```
USES   Peer_Interface (peers) with prefix peer_
```

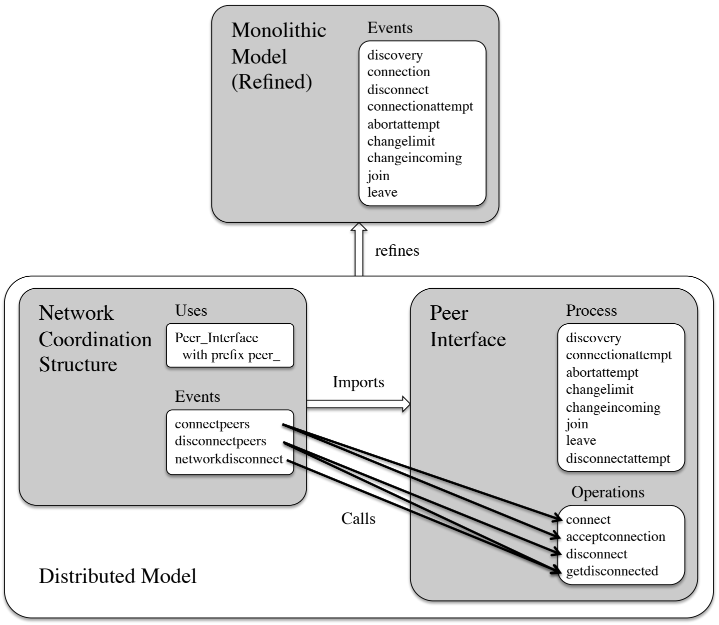

Fig. 4. Decomposition refinement

We previously added a guard specifying that a *connectionattempt* must be done before a *connection*. Here, the *connectionattempt* is internal to the peer, and the network coordination structure has an event *connectpeers*. Given two different online peers s and t, who are not connected to each other, and where s has made a connection attempt to t and t can accept an incoming connection, the operation *acceptconnection* of peer t is called with the argument s, and likewise the operation *connect* of peer s is called with the argument t.

```
EVENT  connectpeers ≙
    any
        $s\ t$
    where
        grd1 :  $s \in peers \land t \in peers \land s \neq t$
        grd2 :  $t \in peer\_connectionattempt(s) \land peer\_acceptincoming(t) = TRUE$
        grd3 :  $t \notin peer\_connection(s) \land s \notin peer\_connection(t)$
        grd4 :  $peer\_isonline(s) = TRUE \land peer\_isonline(t) = TRUE$
        grd5 :  $peer\_connections(t) < peer\_connectionlimit(t)$
    then
        act1 :  $void1 := peer\_acceptconnection(t)(s)$
        act2 :  $void2 := peer\_connect(s)(t)$
    end
```

The variables *void1* and *void2* used here are of the type *VOID*, which is used when an operation call has no return value.

There is a difference between the *acceptconnection* and *connect* operations of the peer interface, in that the former is to be called on a peer that has not made a connection attempt, while the latter is to be called on a peer who has made one. This means that among the preconditions for the *acceptconnection* operation is that the peer must accept incoming connections and must not have reached its connection limit.

```
OPERATION  $acceptconnection \mathrel{\widehat{=}}$
    any
        $dest$
    pre
        pre1 :  $dest \in peers \wedge dest \notin connection \wedge dest \notin connectionattempt$
        pre2 :  $connections < connectionlimit$
        pre3 :  $acceptincoming = TRUE \wedge isonline = TRUE$
    return
        $void$
    post
        post1 :  $connection' = connection \cup \{dest\}$
        post2 :  $connections' = connections + 1$
        post3 :  $void' :\in VOID$
    end
```

For the peer receiving the *connect* operation call the amount of connections was already increased when making the connection attempt, and therefore should not be increased here. However, as a peer is added to the set of connections, it must also be removed from the set of connection attempts.

```
OPERATION  $connect \mathrel{\widehat{=}}$
    any
        $dest$
    pre
        pre1 :  $dest \in peers \wedge dest \notin connection \wedge dest \in connectionattempt$
        pre2 :  $isonline = TRUE$
    return
        $void$
    post
        post1 :  $connection' = connection \cup \{dest\}$
        post2 :  $connectionattempt' = connectionattempt \setminus \{dest\}$
        post3 :  $void' :\in VOID$
    end
```

In our monolithic model, the *disconnect* event simply disconnected two peers that were connected. However, we note that in this refinement we need to separate disconnection into two cases; the first of which concerns a peer actively wanting to disconnect from another peer, and another case when the disconnection happens without the intent of any of the peers involved. In the first case, we will handle it similarly to the connection process. In the peer interface, we specify a new process, *disconnectattempt*, which modifies a variable that will be read by the network coordination machine. When the prerequisites are fulfilled, i.e., when two distinct peers are connected and one of them has made a disconnection attempt concerning the other, the event *disconnectpeers* in the machine then calls the *disconnect* operation on the originating peer and *getdisconnected* on the other.

```
EVENT  $disconnectpeers \mathrel{\widehat{=}}$
    any
        $p\ r$
    where
        grd1 :  $p \in peers \wedge r \in peers \wedge p \neq r$
        grd2 :  $r \in peer\_connection(p) \wedge p \in peer\_connection(r)$
        grd3 :  $r \in peer\_disconnectionattempt(p)$
    then
        act1 :  $void1 := peer\_getdisconnected(r)(p)$
        act2 :  $void2 := peer\_disconnect(p)(r)$
    end
```

In the peer interface, the two operations *getdisconnected* and *disconnect* are very similar. In the first, the peer must remove the connection to a specific peer for which no “disconnection attempt” has been created, and decrease the number of total connections.

```
OPERATION  $getdisconnected \mathrel{\widehat{=}}$
    any
        $dest$
    pre
        pre1 :  $dest \in peers \land dest \in connection \land dest \notin disconnectionattempt$
        pre2 :  $connections > 0$
    return
        $void$
    post
        post1 :  $connection' = connection \setminus \{dest\}$
        post2 :  $connections' = connections - 1$
        post3 :  $void' :\in VOID$
    end
```

For the *disconnect* operation to be enabled, there must be a "disconnection attempt", but otherwise the preconditions are the same as in the *getdisconnected* operation. The postconditions are also identical to the previously described operation, with the addition that the "disconnection attempt" must also be removed.

```
OPERATION  $disconnect \mathrel{\widehat{=}}$
    any
        $dest$
    pre
        pre1 :  $dest \in peers \land dest \in connection \land dest \in disconnectionattempt$
        pre2 :  $connections > 0$
    return
        $void$
    post
        post1 :  $connection' = connection \setminus \{dest\}$
        post2 :  $connections' = connections - 1$
        post3 :  $void' :\in VOID$
        post4 :  $disconnectionattempt' = disconnectionattempt \setminus \{dest\}$
    end
```

As we mentioned, two peers can get disconnected not only by their own intent but also because of external factors. We model this in the network structure machine with the event *networkdisconnect*. This event simply calls the operation *getdisconnected* on each of the two peers, with the other peer as argument, with the prerequisite that the peers must be connected to each other but not have tried to disconnect of their own intent.

```
EVENT  $networkdisconnect \mathrel{\widehat{=}}$
    any
        $u \ v$
    where
        grd1 :  $u \in peers \land v \in peers \land u \neq v$
        grd2 :  $v \in peer\_connection(u) \land u \in peer\_connection(v)$
        grd3 :  $v \notin peer\_disconnectattempt(u) \land u \notin peer\_disconnectattempt(v)$
    then
        act1 :  $void1 := peer\_getdisconnected(u)(v)$
        act2 :  $void2 := peer\_getdisconnected(v)(u)$
    end
```

The intended goal when creating any formal model such as ours is to be able to prove various properties in the system being modelled. For our monolithic model, all generated proof obligations can be easily discharged using the proving environment of the Rodin platform tool [11]. As the Modularisation plugin includes proof generation and proving support for its extensions to the Event-B language [13], many of the properties that we can prove in the original monolithic model we can also prove in the distributed model. We put forward an example of the transformation of a property to prove from the monolithic to the distributed model in the following section.

Using this modularisation technique to do decomposition refinement increases the complexity of the model, which makes proving more difficult. This applies equally to the automatic proof obligation discharging and interactive proving in the Rodin platform tool. As the tool and the Modularisation plugin evolves, we hope that it will enable us to develop our models further than what is currently possible.

5 Discussion

In this section we summarise the contributions of this paper.

First, we propose a (stepwise developed) monolithic model for inter-peer relations in a peer-to-peer network. This model has a (simple) state consisting of the values of the variables and a set of events that can all access and modify the state. Due to the high level of abstraction, we can formulate and prove various properties about our model. For instance, we have an invariant stating that any peer that is connected to another peer, i.e., has a "connection" relation to it, cannot have a "connection attempt" relation to the same peer, put forward below. Coordination between peers is centralised and endogenous, for instance the event *connectionattempt* coordinates the establishment of a "pre-connection" relation and the event *connection* coordinates the establishment of a real "connection" when the proper conditions for it are met.

$$\forall p, r \cdot (\{p \mapsto r\} \in \mathit{connection}) \Rightarrow (\{p \mapsto r\} \notin \mathit{connectionattempt}) \tag{1}$$

Second, we refine the monolithic model into a distributed one, in which we separate the coordination between peers from the internal actions (computation) of the peers. Coordination is now exogenous, modelled by the events *connectpeers*, *disconnectpeers*, and *networkdisconnect*. The properties proven in the monolithic model evolve as well, as exemplified in the following. In the peer interface of the distributed model, we are modelling the interface of one peer, and therefore that peer does not need to be included. Thus, the corresponding invariant in the peer interface states that each peer that we have a connection to must not be in the set of connection attempts.

$$\forall p \cdot p \in \mathit{peers} \land p \in \mathit{connection} \Rightarrow p \notin \mathit{connectionattempt} \tag{2}$$

In both cases, it is of course also trivial to prove the inverse implication, i.e., that a connection attempt between two peers implies that there is no existing connection between the two.

We note that the coordination in the distributed model is rather sophisticated. The coordinator (the network coordination structure) only reads the value of the coordinated peer state (via external variables such as *peer_connection(u)* in event *networkdisconnect*). The peer state is only modified via the nodes' own actions, as described in the operation *getdisconnected*. We can also argue for the coordination paradigm displayed by our modelling to be of a mixed nature. On one hand, the external variables of the peers model a distributed (tuple) space; the coordinator only acts based on reading this space, hence a data-driven coordination. On the other hand, the execution of the coordination actions is not performed directly on the data, but via procedure calls mechanisms, hence, a control-oriented coordination model.

6 Conclusions

Using the *refinement* approach, a system can be described at different levels of abstraction, and the consistency in and between levels can be proved mathematically. With the aim of modelling and analysing a whole, fully featured peer-to-peer media distribution system, we have used Event-B to model inter-peer relations in a BitTorrent-like peer-to-peer network. We have started from an abstract specification and stepwise introduced functionality so that the proving effort remains reasonable. For instance, we could have introduced the *join* and *leave* events already in the first model; however, this would have generated unnecessary proving at an abstract level.

Our focus has been on creating a model of a peer-to-peer system in a way that allows it to be reused and extended for different protocol additions, while keeping the reliability of the system intact. This gives us a foundation from which we can develop a well behaving and scalable peer-to-peer media distribution system. Our goal is to have all the parts, from the network structure up to the content playback, formally modelled and verified. We have previously modelled different parts of such a system, including algorithms for acquiring pieces of media content [22,23] and parts of a video decoding process [19].

A general strategy of a distributed system development in Event-B is to start from an abstract centralised specification and incrementally augment it with design-specific details. When a suitable level of details is achieved, certain events of the specification are replaced by the calls of interface operations and variables are distributed across modules [12]. As a result, a monolithic specification is decomposed into separate modules. Since decomposition is a special kind of refinement, such a model transformation is also correctness-preserving. Therefore, refinement allows us to efficiently cope with complexity of distributed systems verification and gradually derive an implementation with the desired properties and behaviour [2].

With respect to proving properties about models, our strategy is very useful: we formulate and prove properties for the monolithic model and then we develop the distributed model from the monolithic one so that the properties remain valid. This is however not new, as it has been proposed in a number of earlier works, for instance in [5]. With respect to the coordination paradigm, we consider that modularisation in Event-B provides a very interesting methodology for emphasising the separation of the coordination features from the computation ones. This is especially useful in the context of the Rodin tool platform [11] that can significantly improve the property proving effort and thus puts forward our approach to coordination as a practical one.

As future work, we plan to develop the peer-to-peer networking models into an Event-B theory. This means that we can then model specific peer-to-peer networks simply by instantiating them from the theory, much like declaring data types. Hence, we envision a language construct for modern network architectures. With this, we stress once more the reuse potential of our proposal.

Acknowledgements. We would like to thank Alexei Iliasov for helping us to better understand the Modularisation plugin.

References

1. Abrial, J.R.: The B-Book: Assigning Programs to Meanings. Cambridge University Press (1996)
2. Abrial, J.R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press (2010)
3. Abrial, J.R., Butler, M., Hallerstede, S., Hoang, T.S., Mehta, F., Voisin, L.: Rodin: An Open Toolset for Modelling and Reasoning in Event-B. International Journal on Software Tools for Technology Transfer (STTT) 12(6), 447–466 (2010)
4. Abrial, J.R., Butler, M., Hallerstede, S., Voisin, L.: An Open Extensible Tool Environment for Event-B. Lecture Notes in Computer Science 4260, 588–605 (2006)
5. Back, R., Kurki-Suonio, R.: Decentralization of Process Nets with Centralized Control. In: Proceedings of the 2nd ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing. pp. 131–142 (1983)
6. Belkin Play N600 HD Wireless Dual-Band N+ Router F7D8301. `http://www.belkin.com/IWCatProductPage.process?Product_Id=522112` (Accessed 04/2012)
7. Carriero, N., Gelernter, D.: Data Parallelism and Linda. In: Languages and Compilers for Parallel Computing, 5th International Workshop, New Haven, Connecticut, USA, August 3-5, 1992, Proceedings. Lecture Notes in Computer Science, vol. 757, pp. 145–159. Springer (1993)
8. Cohen, B.: Incentives Build Robustness in BitTorrent. In: 1st Workshop on Economics of Peer-to-Peer Systems (June 2003)
9. Cohen, B.: The BitTorrent Protocol Specification. `http://www.bittorrent.org/beps/bep_0003.html` (Accessed 04/2012) (January 2008)
10. D'Acunto, L., Meulpolder, M., Rahman, R., Pouwelse, J., Sips, H.: Modeling and Analyzing the Effects of Firewalls and NATs in P2P Swarming Systems. In: IEEE International Symposium on Parallel & Distributed Processing, Workshops and PhD Forum (IPDPSW) (2010)
11. Event-B and the Rodin Platform. `http://www.event-b.org/` (Accessed 04/2012)
12. Iliasov, A., Troubitsyna, E., Laibinis, L., Romanovsky, A.: Formal Derivation of a Distributed Program in Event-B. In: ICFEM'11. pp. 420–436. LNCS (2011)
13. Iliasov, A., Troubitsyna, E., Laibinis, L., Romanovsky, A., Varpaaniemi, K., Ilic, D., Latvala, T.: Supporting Reuse in Event-B Development: Modularisation Approach. In: Frappier, M., Glässer, U., Khurshid, S., Laleau, R., Reeves, S. (eds.) Abstract State Machines, Alloy, B and Z. Lecture Notes in Computer Science, vol. 5977, pp. 174–188 (2010)
14. Iliofotou, M., Siganos, G., Yang, X., Rodriguez, P.: Comparing BitTorrent Clients in the Wild: The Case of Download Speed. In: Freedman, M.J., Krishnamurthy, A. (eds.) Proceedings of the 9th International Workshop on Peer-to-Peer Systems (IPTPS '10). USENIX (April 2010)

15. Kamali, M., Laibinis, L., Petre, L., Sere, K.: Self-Recovering Sensor-Actor Networks. In: Mousavi, M., Salan, G. (eds.) Proceedings of the Ninth International Workshop on the Foundations of Coordination Languages and Software Architectures (FOCLASA 2010). vol. 30, pp. 47–61. EPTCS (2010)
16. Kemper, S.: Compositional construction of real-time dataflow networks. In: Clarke, D., Agha, G. (eds.) Coordination Models and Languages. Lecture Notes in Computer Science, vol. 6116, pp. 92–106. Springer Berlin / Heidelberg (2010)
17. Loewenstern, A.: DHT Protocol. `http://www.bittorrent.org/beps/bep_0005.html` (Accessed 04/2012) (2008)
18. Lombide Carreton, A., D'Hondt, T.: A Hybrid Visual Dataflow Language for Coordination in Mobile Ad Hoc Networks. In: Clarke, D., Agha, G. (eds.) Coordination Models and Languages, Lecture Notes in Computer Science, vol. 6116, pp. 76–91. Springer Berlin / Heidelberg (2010)
19. Lumme, K., Petre, L., Sandvik, P., Sere, K.: Towards Dependable H.264 Decoding. In: Ahmed, N., Quercia, D., Jensen, C.D. (eds.) Workshop Proceedings of the Fifth IFIP WG 11.11 International Conference on Trust Management (IFIPTM 2011). pp. 325–337. Technical University of Denmark (June 2011)
20. Petre, L., Sandvik, P., Sere, K.: A Modular Approach to Formal Modelling of Peer-to-Peer Networks. Tech. Rep. 1039, Turku Centre for Computer Science (TUCS) (2012)
21. RODIN Modularisation Plug-in. Documentation at `http://wiki.event-b.org/index.php/Modularisation_Plug-in` (Accessed 04/2012)
22. Sandvik, P., Neovius, M.: The Distance-Availability Weighted Piece Selection Method for BitTorrent: A BitTorrent Piece Selection Method for On-Demand Streaming. In: Liotta, A., Antonopoulos, N., Exarchakos, G., Hara, T. (eds.) Proceedings of The First International Conference on Advances in P2P Systems (AP2PS 2009). pp. 198–202. IEEE Computer Society (October 2009)
23. Sandvik, P., Sere, K.: Formal Analysis and Verification of Peer-to-Peer Node Behaviour. In: Liotta, A., Antonopoulos, N., Di Fatta, G., Hara, T., Vu, Q.H. (eds.) The Third International Conference on Advances in P2P Systems (AP2PS 2011). pp. 47–52. IARIA (November 2011)
24. Schulze, H., Mochalski, K.: Ipoque Internet Study 2008/2009. `http://www.ipoque.com/en/resources/internet-studies` (Accessed 04/2012)
25. Tarau, P.: Coordination and Concurrency in Multi-Engine Prolog. In: De Meuter, W., Roman, G.C. (eds.) Proceedings of the 13th International Conference on Coordination Models and Languages (COORDINATION'11). Lecture Notes in Computer Science, vol. 6721, pp. 157–171. Springer-Verlag (2011)
26. Vestel to Launch the First Bittorrent Certified Smart TV. `http://www.bittorrent.com/company/about/vestel_to_launch_the_first_bittorrent_certified_smart_tv` (Accessed 04/2012)
27. Waldén, M., Sere, K.: Reasoning About Action Systems Using the B-Method. Formal Methods in Systems Design 13, 5–35 (1998)
28. Yan, L.: A Formal Architectural Model for Peer-to-Peer Systems. In: Shen, X., Yu, H., Buford, J., Akon, M. (eds.) Handbook of Peer-to-Peer Networking 2010 Part 12, pp. 1295–1314. Springer US (2010)
29. Yan, L., Ni, J.: Building a Formal Framework for Mobile Ad Hoc Computing. In: Bubak, M., van Albada, G.D., Sloot, P.M.A., Dongarra, J. (eds.) International Conference on Computational Science (ICCS 2004). Lecture Notes in Computer Science, vol. 3036, pp. 619–622. Springer (June 2004)