Relating Layered Queueing Networks and Process Algebra Models

Mirco Tribastone
School of Informatics
The University of Edinburgh
Edinburgh, United Kingdom
mtribast@inf.ed.ac.uk

ABSTRACT

This paper presents a process-algebraic interpretation of the Layered Queueing Network model. The semantics of layered multi-class servers, resource contention, multiplicity of threads and processors are mapped into a model described in the stochastic process algebra PEPA. The accuracy of the translation is validated through a case study of a distributed computer system and the numerical results are used to discuss the relative strengths and weaknesses of the different forms of analysis available in both approaches, i.e., simulation, mean-value analysis, and differential approximation.

Categories and Subject Descriptors
I.6.5 [Simulation and Modeling]: Model Development—Modeling methodologies; D.2.8 [Software Engineering]: Metrics—Performance measures

General Terms
Performance

1. INTRODUCTION

One of the main advantages of queueing networks for the performance evaluation of software systems is the availability of computationally inexpensive analysis techniques which scale well with increasing sizes of the system under study. Queueing models can be subjected to mean-value analysis (MVA) [19] and more efficient variants thereof [4, 7], which use the arrival theorem and Little’s law to compute accurate estimates of average steady-state performance metrics (e.g. throughput, utilisation, and response time), many orders of magnitude faster than simulative approaches.

The widespread acceptance of queueing theory in the software performance evaluation community has fostered a large body of research on extending this theory to capture the dynamics which naturally emerge from complex distributed software systems. A fundamental contribution of this line of inquiry is the notion of layered servers. In Woodside’s Stochastic Rendezvous Network model servers may also act as clients for services offered by other lower-level servers. In addition, a service may consist of two or more phases, in which the first phase models the time between the request and the corresponding reply to the client, whereas the subsequent ones describe server-side independent computation [29]. Rolia’s Method of Layers proposes a similar approach for the description of software/hardware models with layers and resource contention [20]. The Layered Queueing Network (LQN) model has been shown to include all these features and support further extensions, including activity graphs for sequence, conditional (probabilistic) branching, fork/join semantics, and quorum consensus synchronisation [9].

As an alternative approach, the distributed nature of the computation can be captured with process-algebraic techniques. Here, models are constructed as compositions of agents which are capable of carrying out activities either autonomously or in cooperation with other agents (e.g., [17, 16]). This form of cooperation models a synchronisation point which can be used to express communication. Process algebras were originally developed as formal specification techniques in which time was not explicitly considered, thus serving as useful tools for the verification of qualitative properties only. More recently, stochastic extensions have provided mappings onto continuous-time Markov chains for performance evaluation [2, 12, 14].

The present work was inspired by [13], in which the authors present a comparison between the Method of Layers model and stochastic process algebras. The two techniques are presented as achieving orthogonal goals: on the one hand, layered queues have the advantage of relatively low computational cost, but performance results are limited to mean performance indices; on the other hand, process algebras can provide a richer set of qualitative and quantitative results. However, the computational cost is usually high because of the discrete nature of the underlying mathematics, which is likely to cause combinatorial growth of the state space with increasing number of components in the system (state-space explosion problem).

This paper revisits this relationship from a different perspective, discussing the benefits of a process-algebraic interpretation of LQN models in light of recent theoretical developments for the scalability of the analysis of process algebra models. In particular, the focus is on the process algebra PEPA [14], in which the problem of state explosion has been tackled by providing a semantics onto an efficient stochastic simulation model [3], and more radically with a continuous-
state mapping based on ordinary differential equations [15].

Section 2 gives an overview of the LQN model and PEPA, emphasising the main notions which will be used in the remainder of the paper. Section 3 presents a methodology for mapping LQN elements into PEPA processes, covering many important features such as multiplicity of classes of servers, multithreaded and multiprocessor computation, synchronicity of service requests, and the fork/join paradigm for concurrent behaviour. It also discusses how to obtain corresponding indices in the PEPA model for throughput and utilisation. The overall methodology—which is general and thus can be implemented for automatic translations—is practically applied to a case study of a distributed system. In Section 4, this model is used to validate the translation and compare all forms of analysis available for the two techniques on the basis of accuracy, computational cost, and richness of the result set. Finally, Section 5 concludes the paper and gives directions for future work.

2. BACKGROUND

2.1 Layered Queueing Networks

This section gives an informal overview of the LQN model by means of a running example. The reader is referred to [9] (and the rich bibliography therein) for a more detailed treatment. Figure 1 shows a LQN of a distributed application which features all of the elements considered in this paper. Servers (called tasks) are drawn as stacked parallelograms and its multiplicity is indicated within angular brackets alongside the task’s name. For instance, File Server <1> denotes one single thread of execution for the file server. A task is deployed onto a processor, depicted as a circle connected to the task. Concurrency levels for processors are denoted similarly to tasks.

Distinct kinds of services (called entries) exposed by a task are represented by small parallelograms drawn inside the task. Each entry is associated with an execution graph consisting of atomic units of computation called activities, drawn as rectangles. Activities are arranged through operators for sequencing (directed arrows), conditional branching/merging (small circle with the + symbol) and fork/join synchronisation (small circle with the & symbol). Each activity is characterised by a service time demand on the processor with which the task is associated, indicated within square brackets. For the sake of graphical convenience, execution graphs which consist of a single activity are not explicitly drawn, and the activity’s execution demand is directly shown within the associated entry. In Figure 1 only the execution graphs of entry visit and buy are drawn. The former models an activity which accesses some cached information, after which it performs an internal activity with probability 0.95 or a more expensive external activity with probability 0.05. In buy, after the activity prepare is performed the two activities packaging and shipping are executed in parallel. When they both finish, display is executed.

Layering of services is modelled by means of requests made from an activity to an entry in another task in the network. Requests are indicated by directed arrows and may be of two kinds: synchronous, with closed arrowheads, and asynchronous, with open arrowheads. Each request is labelled with a number between parentheses, which gives the number of requests per execution. This can be interpreted deterministically or as the mean of a geometric distribution. The total number of requests performed by an activity determines the distribution of its execution demand. The total demand is divided into slices whose duration is drawn from independent exponential distributions with mean equal to the ratio between total execution demand and total number of requests. The execution of one slice is interposed between successive requests to other entries. Reference tasks are tasks which do not accept requests and they are used to model system workload.

For entries which accept synchronous requests, their overall behaviour may be subdivided into two phases. The first phase models the computation carried out from the receipt of the request until the reply to the caller. Such a reply is denoted as a dashed arrow pointing to the activity’s entry. All such activities in the execution graph that follow the replying entry are part of the second phase, indicating an autonomous continuation during which the caller is not blocked. Execution graphs consisting of two activities such that each represents the behaviour of one phase can be conveniently drawn in a compact form, as illustrated by write in Figure 1. The execution demand for each phase is drawn inside the entry within square brackets. The requests from multi-phase entries are labelled with pairs, in which the i-th element represents the number of requests made by the activity in the i-th phase.

2.2 PEPA

A PEPA model consists of a composition of entities which can perform actions sequentially (sequential components). Actions may be performed autonomously (independent ac-
Prefix \((\alpha, r) \cdot P\) constitutes the atomic unit of computation of a PEPA model. It is a sequential component which may perform an action of type \(\alpha\), subsequently behaving as \(P\), which is said to be a local derivative (or local state) of the component. The duration of the action is drawn from an exponential distribution with mean \(1/r\) time units.

Choice \(P + Q\) indicates that the sequential component may behave as \(P\) or \(Q\). For instance \((\alpha, r) \cdot P + (\beta, s) \cdot Q\) is said to enable actions \(\alpha\) and \(\beta\), which are executed with probabilities \(r/(r+s)\) and \(s/(r+s)\), respectively. Choice between distinct action types will be used to model different entries within the same task. With a slight abuse, a PEPA choice between prefixes will be described using the following sigma notation:

\[
\sum \{ (\alpha, r) \cdot P \mid \mathcal{P}(\alpha, r, P) \},
\]

where \(\mathcal{P}(\alpha, r, P)\) is a predicate of the choice’s constituting prefixes, e.g.,

\[
\sum \{ (\alpha, i \cdot r).P_i \mid 1 \leq i \leq 2 \} \overset{\text{def}}{=} (\alpha, r).P_1 + (\alpha, 2r).P_2
\]

Constant \(A \overset{\text{def}}{=} P\) is used to model cyclic behaviour. For instance, \(A \overset{\text{def}}{=} (\alpha, r). (\beta, s). A\) is a sequential component with two local derivatives which performs sequences of \(\alpha\)- and \(\beta\)-activities forever.

Cooperation \(P \overset{\text{sync}}{\parallel} Q\) is the synchronisation operator of the language. The processes \(P\) and \(Q\) are required to synchronise over the action types in the set \(L\). All the other actions are performed autonomously. For instance \((\alpha, r). (\beta, s). P \overset{\text{sync}}{\parallel} (\alpha, t). (\gamma, u). Q\) is a cooperation between two sequential components which may perform a shared activity of type \(\alpha\), subsequently behaving as \((\beta, s). P \overset{\text{sync}}{\parallel} (\gamma, u). Q\). Then actions \(\beta\) and \(\gamma\) are carried autonomously. By contrast, in the cooperation \((\alpha, r). P \overset{\text{async}}{\parallel} (\beta, s). Q\) the process \((\alpha, r). P\) does not progress because \(\alpha\) is not available in the right hand side of the cooperation. The set of all shared action types between \(P\) and \(Q\) will be denoted by the symbol \(s\). Cooperation will be used in the translation of LQNs into PEPA for two main modelling situations: (i) the definition of synchronisation point in forks and joins; (ii) passing the focus of control from one sequential component to another, e.g., a request from an activity to an entry.

Hiding \(P/L\) renames all the action types of \(P\) as silent actions, indicated by the action type \(\tau\). Hiding is not used in the remainder of this paper.

### 3. PEPA INTERPRETATION OF LQNS

The main rationale behind the PEPA interpretation of LQNs presented in this section is to exploit the inherent concurrent behaviour of replicated tasks and processors, and model those as copies of identical sequential components in the process-algebra model. Thus, if \(T\) is the sequential component which describes the behaviour of a task thread, then the whole server is described as \(T \parallel T \parallel \cdots \parallel T\),

where \(N\) is the multiplicity of the server in the LQN model. The main benefit in using replicated components is that the model can be subjected to symmetry reduction, leading to potentially much smaller underlying Markov chains [11]. Furthermore, when interpreted against the continuous-state semantics, the size and structure of the differential equation is not dependent upon the actual population levels of the system.

The interpretation of each LQN element is now discussed.
in more detail. Table 1 summarises the notation used in this section.

### 3.1 Processor

The template for the translation of a single processor \( p \) is illustrated in Figure 2, showing a cyclic two-state sequential component. The first state \( \text{Proc}_p \) models an activity which grants exclusive access to the processor. The rate \( \nu \) for this action is assumed to be much faster than any other activity in the system. The impact of this rate on the performance results will be examined in Section 4.1.

The second state \( \text{Exec}_p \) enables all the actions corresponding to the activities which are executed on \( p \), by means of the delay operator. Each activity phase is mapped onto a distinct action type in PEP A and the rate of execution reflects the fragmentation of the computation into slices. For any activity \( a \), the rate of execution of a slice is denoted by \( s(a) \) and it is equal to (cfr. process \( \text{Exec}_p \) in Figure 2):

\[
s(a) = \begin{cases} 
(N(a) + 1) \cdot \text{dem}(a)^{-1} & \text{if } \text{dem}(a) > 0 \\
0 & \text{if } \text{dem}(a) = 0
\end{cases}
\]

Notice that this interpretation produces a concise description for a processor, whose size is not dependent upon the distinct classes of service enabled. For example, the translation of a \( \text{PFileServer} \) is shown in Figure 3.

### 3.2 Activity and Request

An LQN activity subsumes a sequence of PEA prefixes, whose length depends upon the number of outgoing requests and their synchronicity. A synchronous call is modelled with a sequence of two prefixes which model the request and the reply. The PEA action type for the request has the form \( \text{request}_{a,e} \), where \( a \) is the activity from which the request originates and \( e \) is the entry called by \( a \). Similarly, the action type for the reply has the form \( \text{reply}_{a,e} \). An asynchronous call is represented with a single prefix of type \( \text{request}_{a,e} \).

The PEA process corresponding to the LQN activity interposes executions of slices of \( a \) between requests. The rates for requests and replies are here set to \( \nu \), i.e., it is assumed that the delay for message exchange is negligible with respect to the execution demands on the processors. With respect to the LQN interpretation, this means that the rate of transition of jobs between queues is very fast. The following snippets of PEA descriptions will be useful for the translation of an activity:

\[
\text{Act}^a_{0} \equiv \begin{cases} 
\text{acquire}^a_{act^{-1}(a)}, \nu \cdot (a, s(a)) & \text{if } s(a) > 0 \\
\epsilon & \text{if } s(a) = 0
\end{cases}
\]

\[
\text{Act}^a_{1} \equiv \begin{cases} 
\text{Sync}_{a,e_1}, \ldots, \text{Sync}_{a,e_n}, \text{Act}^a_{i+2} & \text{if } k_i = \text{sync} \\
\text{Async}_{a,e_1}, \ldots, \text{Async}_{a,e_n}, \text{Act}^a_{i+2} & \text{if } k_i = \text{async}
\end{cases}
\]

\[
\text{Act}^a_{i+1} \equiv \begin{cases} 
\text{rep}(a_e)^{\nu} \cdot \text{req}(a_e)_{i+1} \rightarrow (a_e)^{\nu} & \text{if } k_i = \text{sync} \\
\text{rep}(a_e)^{\nu} \cdot \text{req}(a_e)_{i+1} \rightarrow (a_e)^{\nu} & \text{if } k_i = \text{async}
\end{cases}
\]

Figure 4: Translation of an LQN Activity.

where \( \text{Acq}_a \) models the access to the processor and the execution of a slice of activity \( a \). It is an empty string \( \epsilon \) if the activity has no execution demand (with usual properties of concatenations of empty strings with arbitrary PEA definitions). \( \text{Sync}_{a,e} \) and \( \text{Async}_{a,e} \) model the sequences of prefixes for synchronous and asynchronous requests (followed by slice executions), respectively.

The translation of an LQN activity is shown in Figure 4. The first process definition \( \text{Act}^a_{i} \) models the first slice execution. If the activity replies to a synchronous request then the last constant enables all actions which model replies to any other activity in the network making a request to the entry which owns \( a \). The constant \( \text{End}_a \) is left unspecified and it is defined according to the structure of the LQN, as discussed in Section 3.3. As a concrete application, the translation of write is given in Figure 5. Recalling the semantics of implicit activity invocation, write represents two distinct activities, here denoted by \( \text{write}^1 \) and \( \text{write}^2 \). Activity \( \text{write}^1 \) does not make requests to lower-level server but it replies to requests to entry write made by get. Activity \( \text{write}^2 \) is the autonomous continuation which makes two synchronous requests to the entries get and update of task Backup.

### 3.3 Execution Graph

The interpretation of execution graphs follows the rationale behind the translation of UML activity diagrams into PEP A models presented in [22]. (The reader is referred to that paper for a detailed algorithmic description) This section presents a conceptual view of the approach, focussing
The translation of probabilistic branching (called \textit{Probabilistic Branching} in the LQN context) is visually represented by directed arrows connecting two elements of the graph and it can also be implicitly defined by the 

\begin{align*}
\text{Precedence} & \\
\text{Second phase} & \\
\text{First phase} & \\
\text{Probabilistic Branching} & \\
\end{align*}

on the main differences with respect to the original work.

The analogue of a UML action node in the LQN context is an activity, which represents the atomic unit of computation in an execution graph. However, while an action node is translated into a single PEP A prefix, an activity is a sequential component with several local derivatives. Nevertheless, the two representations share the property that they exhibit some form of sequential computation whose collective behaviour for the purposes of the translation can be summarised by two PEP A constants which define the initial and the final state (i.e., \textit{Act}_0 and \textit{End}_0 in the LQN model). Such definitions are modified in order to combine distinct activities according to the semantics of the execution graph.

For activity/execution graphs, the translation algorithm identifies a number of concurrent control flows. Flows are created by means of fork nodes (called \textit{And-Forks} in the LQN model). For each entry there will be at least one flow, called the main flow, which executes the initial activity of the entry’s execution graph. The overall model of an execution graph can be written in the form \( S \in \mathcal{D} \) where \( S \) is an arbitrary PEP A process consisting of the sequential components which model the remaining control flows, called secondary flows.

\begin{align*}
\text{Precedence} & \\
\text{Second phase} & \\
\text{First phase} & \\
\text{Probabilistic Branching} & \\
\end{align*}

The operator of precedence models the behaviour of one activity being executed after the previous one terminates. It is visually represented by directed arrows connecting two elements of the graph and it can also be implicitly defined by second-phase entries. The notion of precedence in PEP A is represented by letting the final state of the preceding element coincide with the initial state of the subsequent one. For instance, the two phases of the entry \textit{write}—represented in Figure 5 as two unrelated sequential components with no notion of precedence relationship—are transformed into a sequence of activities by letting \( \text{EndWrite}' \equiv \text{Write}' \) (recall that \( \text{EndWrite}' \) was left intentionally unspecified for this purpose).

\begin{align*}
\text{Precedence} & \\
\text{Second phase} & \\
\text{First phase} & \\
\text{Probabilistic Branching} & \\
\end{align*}

The translation of probabilistic branching (called \textit{Or-fork} in the LQN model) involves manipulating all of the activities enabled in the final state of its predecessor and retrieving the information about the constant names which define the initial states of all the successors of the node. According to the template for a basic activity in Figure 4, \textit{cache} is translated into a sequential component in the simple form

\begin{align*}
\text{Cache} & \equiv (\text{acquire}_{ps}, \nu).(\text{cache}, 1/0.001).\text{EndCache}.
\end{align*}

Being the predecessor of a branching operator, its last activity \textit{cache} is replaced with a PEP A choice as follows:

\begin{align*}
\text{Prepare}_1 & \equiv (\text{prepare}_{ps}, \nu).(\text{prepare}, 1/0.01).\text{EndPrepare} \\
\text{Pack}_1 & \equiv (\text{prepare}_{ps}, \nu).(\text{pack}, 1/0.03).\text{EndPack} \\
\text{Ship}_1 & \equiv (\text{prepare}_{ps}, \nu).(\text{ship}, 1/0.01).\text{EndShip}
\end{align*}

As with probabilistic branching, the translation of a fork operator takes as input the set of activities enabled by the initial state of the incoming flow and the set of initial states of the spawned flows. Each activity in the former set is prefixed with a \textit{fork} activity, carried out at rate \( \nu \), indicating a negligible rate of spawning new processes. Each state in the latter set is instead modified so as to have \textit{fork} as the first enabled activity. For instance, the PEP A component corresponding to the basic activity \textit{prepare} is modified to become

\begin{align*}
\text{Prepare}_1 & \equiv (\text{acquire}_{ps}, \nu).(\text{prepare}, 1/0.01).\text{ForkPrepare} \\
\text{ForkPrepare} & \equiv (\text{fork}_1, \nu).\text{EndPrepare}
\end{align*}

where the subscript in the action \textit{fork}_1 is used to uniquely assign a type to each fork node in the execution graph, for instance by mapping them into integers. Similarly, \textit{Pack}_1
and $Ship_1$ are prefixed with the $fork_1$, i.e.,

$$Pack_3 \triangleq (fork_1, \nu).(acquire_{ps_3}, \nu).(pack, 1/0.03).EndPack$$

$$Ship_1 \triangleq (fork_1, \nu).(acquire_{ps_1}, \nu).(ship, 1/0.01).EndShip$$

At a join, the algorithm resolves the unspecified final constants of its incoming flows, by making them synchronise over a $join$ activity (performed at rate $\nu$) and subsequently cycle back to the flows’ initial states. In the example,

$$EndPack \triangleq (join_1, \nu).Pack_1$$

$$EndShip \triangleq (join_1, \nu).Ship_1$$

Furthermore, the translation of a join is responsible for resolving the unspecified final constant of the incoming flow at the matching fork, to capture the following semantics: at a fork, the incoming flow of execution spawns as many flows as the number of successors, and it is suspended until all of them have terminated; then, it behaves as the flow corresponding to the outgoing edge at the matching join. In the example, the component

$$Display_i \triangleq (acquire_{ps_i}, \nu).(display, 1/0.001).EndDisplay$$

models the behaviour of the outgoing edge of the join. The unspecified constant $EndPrepare$ is defined as follows:

$$EndPrepare \triangleq (join_1, \nu).Display_1$$

**Overall Model of an Execution Graph**

Finally, the complete model of an execution graph is represented as a composition of all the flows’ sequential components, cooperating over the action types for forking and joining. The definitions of the secondary flows are not modified any further, thus they are instantiated with suitable replication according to the multiplicity of the task to which the execution graph belongs. Instead, the definitions of the main flow will be altered when translating an LQN task, during which its multiplicity will be adjusted. In the example, the PEPA model of the execution graph for buy is:

**Prepare**, **Pack**, **Ship**

The overall behaviour of visit, consisting of a single flow of control is simply represented by the flow’s initial state $Cache_1$. In the remainder of this paper, the overall model of an execution graph will be denoted by the component $Main_e \triangleq Sec_e$, where $Main_e$ is the behaviour of the main flow, without information on its multiplicity, and $Sec_e$ comprises all the secondary flows, with proper multiplicities. The cooperation set $L_e$ consists of fork/join actions in which the main flow is involved throughout its execution. Under conditions of balanced branching (i.e., each flow spawned at a fork eventually joins), only one constant corresponding to the final behaviour of the main flow will be left unspecified — for instance, $Merge$ in visit, $EndDisplay$ in Buy, and $EndWrite$ in write. For an entry $e$ such a constant will be denoted by $Last_e$.

**3.4 Task**

A reference task, here denoted by $t^r$, has the same behaviour as its unique entry, denoted by $e^r$. Instead, a non-reference task is modelled as a PEPA process which initially enables the activities corresponding to the invocations of all its entries, modelled as an initial choice component. When one of such activities is chosen, the process behaves as the initial state of the main flow of the execution graph corresponding to that entry. Then, after all activities in that execution graph are performed, the task component returns to its initial state in which any entry may be executed. The pattern of transformation of a task is shown in Figure 6. For instance, the complete translation of the non-reference task $FileServer$ is given in Figure 7. The entry $read$ starts executing upon the receipt of either of two messages from external or $think$, modelled as two distinct prefixes in the initial choice which behave as the same component $Read_1$ (the actual behaviour of the entry is independent from the originator of the request).

**Reference Task**

$Task_e \triangleq \{ (request_{\text{in}}, e, \nu).Main_e | \exists (c, n, k) \in \text{req}(a): e \in \text{ent}(t), Va \in A \}$

$Last_e \triangleq Task_e$, for each $e \in \text{ent}(t)$

**Non-Reference Task**

$FileServer \triangleq \{ \text{request}_{\text{external}}, \text{read}, \nu).Read_1 + \text{request}_{\text{think}}, \text{read}, \nu).Read_1 + \text{request}_{\text{save}}, \text{write}, \nu).Write_1 \}$

$EndRead \triangleq FileServer$

$Write_1 \triangleq \ldots$

$EndWrite \triangleq FileServer$

**Figure 7: Translation of task FileServer.**

**3.5 Network**

$Comp_r \triangleq Task_r[\text{mtk}(t)] \triangleq L_e \bigcup S_{\text{Sec}_1} \bigcup S_{\text{Sec}_2} \bigcup \ldots \bigcup S_{\text{Sec}_{\text{in}}(t)}$

where $L_e = \bigcup_{t \in \text{ent}(t)} L_{c_t}$, for all $t \in T$

$LQN \triangleq \bigcup_{t \in T} \left\{ \text{Proc}_{p_1} \bigcup \text{Proc}_{p_2} \bigcup \ldots \right\}$

where $\tilde{M} = * - \bigcup_{p \in P} \{ \text{acquire}_p \}$

**Figure 8: Translation of an LQN.**
The complete LQN is represented by a PEPA cooperation which arranges all the components as inferred above and introduces the concurrency levels for the entries’ main flows and the processors. The pattern of translation is shown in Figure 8. The definition $\text{Comp}_t$ describes the overall behaviour of a multithreaded server with multiple entries. The task behaviour $\text{Task}_t$ (subsuming all the main flows of a task’s entries) is instantiated with the concurrency level of the task. It is composed in parallel with a group of components (within curly braces), each collecting the behaviour of a secondary control flow for each entry of the task. The cooperation sets between secondary flows of distinct entries are empty because no form of communication is possible between two entries within the same task—an entry’s activity may only request service from another task of the network. Conversely, $\text{Task}_t$ is composed with all its secondary flows over a cooperation set which includes all the fork/join action types in which the main flow of any task’s entry is involved.

The definitions $\text{Comp}_t$ are combined together using cooperation sets which can be denoted by the same expression $\hat{M}$. However, notice that the actual instantiations are all different because of the dependence of the set $\hat{M}$ upon the operands of the cooperation. In fact, it is possible to show that all such sets are pairwise disjoint. Observe that, by construction, all the acquire action types are not contained in the sets $\hat{M}$. Any pair of components of type $\text{Comp}_t$ does not exhibit the same action type for the execution of a basic activity, since each activity belongs to only one task. The same fork/join action type cannot be exhibited because these activities are executed within the same task, and distinct fork/join nodes give rise to distinct action types in the PEPA model. Thus, the only potential elements of $\hat{M}$ are the action types for message exchange request$_{a,e}$ and reply$_{a,e}$. The fact that sets with such action types are pairwise disjoint follows immediately from the uniqueness of activity and entry names in the LQN and can be proven by structural induction.

For an arbitrary composition of three components $\text{Comp}_1$, i.e.,

$$\text{Comp}_1 \otimes \text{Comp}_2 \otimes \text{Comp}_3,$$

component $\text{Comp}_1$ may enable request/reply actions with subscripts $(a’,e’)$, $(a'',e’’)$, $\ldots$, where $e’,e’’$ $\in$ ent($t_1$) and $a,b,\ldots$ are basic activities. If some action with subscript $(a,e)$ was present in both cooperation sets then it would mean that both $\text{Comp}_2$ and $\text{Comp}_3$ can perform the same basic activity $a$, which is a contradiction. Then, assuming that the property holds for a cooperation among $n > 3$ components

$$\text{Comp}_1 \otimes \text{Comp}_2 \otimes \text{Comp}_3 \otimes \text{Comp}_4 \otimes \text{Comp}_5 \otimes \ldots \otimes \text{Comp}_n,$$

it suffices to prove that the cooperation set $\hat{M}$ in position $\text{Comp}_i$ is disjoint from the cooperation set $\text{Comp}_j$, for all $1 \leq i \leq n - 1$. Suppose that for some $i \text{Comp}_i \otimes \text{Comp}_{i+1}$ has some action in common with the set in $\text{Comp}_j \otimes \text{Comp}_{j+1}$. This implies that the action must be a request/reply action with subscript $(a,e)$, $e \in \text{ent}(t_{n+1})$, because it belongs to the set $\text{Comp}_n \otimes \text{Comp}_{n+1}$, and that $e \in \text{ent}(t_{n+1})$, which is a contradiction because $i + 1 \neq n + 1$ but one entry must belong to only one task. This property is of crucial importance because it guarantees that at most two distinct components $\text{Comp}_t$ synchronize for message exchange.

The group of task components is finally combined with the group of processors, each taken with its own multiplicity. Processors do not cooperate with each other because any execution slice must be performed on a single processor. However, the cooperation set between all task components and all processors records the fact that any task may be deployed on any processor, but the actual processor $p$ which executes a given activity $a$ will be the only one which exhibits $a$ in its state $\text{Exec}_p$ (cfr. Figure 2).

The complete PEPA model for the LQN in Figure 1 is shown in Appendix A.

### 3.6 Performance Measures

This section is concerned with establishing a relationship between the utilisation and throughput, as computed from the solution of the LQN model, and the corresponding performance results which can be obtained from the analysis of the corresponding PEPA model. Such metrics will be used in Section 4 to quantitatively assess the soundness of the translation proposed in this paper.

**Utilisation**

In the LQN model, utilisation is a performance measure which indicates the mean number of busy processors at equilibrium. Hence, it is a figure between zero and the multiplicity of a processor. More fine-grained results can be obtained by considering the distinct contributions from each of the activities which run on the processor, the total utilisation figure being the sum across all such contributions.

In the PEPA model, the total utilisation for a processor $p$ may be obtained by the mean number of components which are in state $\text{Exec}_p$ (cfr. Figure 2). However, this information alone is not sufficient to obtain the contributions from each of the activities. In order to do so, it is necessary to compute the expectations of all the sequential components of an activity which perform execution slices on the processor. Then, the contribution for an activity is given as the summation across all such expectations. Clearly, if an activity has two phases, the total contribution is the sum of the contributions of each phase. For instance, the utilisation of processor $\text{PFileServer}$ due to the execution of write is obtained by inspection of the sequential components in Figure 5. The utilisation during the first phase is obtained as the expectation of the number of sequential components which behave as write$_{0}$,Write$_{2}'$, whereas the utilisation during the second phase is the sum of the expectation of the following three sequential components: write$_2$, write$_2'$,Write$_2''$, Write$_2''$, and (write$_2$, 3/0.04).EndWrite$_{2}''$. 

**Throughput**

The notion of throughput in the LQN model is associated with a basic activity and it indicates the average number of executions per unit of time. If the activity makes further requests to other entries, the time taken by the other servers is also taken into account. Conversely, the traditional definition of throughput for PEPA is related to an action type (e.g., [22]), implying that it is given for the execution.
of a single slice of an activity. Therefore, the throughput in the sense of the LQN model may be obtained by dividing the throughput per execution slice by the total number of slices for a basic activity. If the activity has two phases, the throughputs of each phase are summed and then divided by the total number of execution slices. For instance, the throughput of write in the sense of the LQN model corresponds to the sum of the throughputs of actions write1 and write2, divided by four. As a special case, the LQN throughput of an activity corresponds to the PEPA throughput of the associated action type if the activity makes no requests to other entries.

4. VALIDATION

The model in Figure 1 was used to conduct a validation study on the quality of the translation. The notion of accuracy used throughout this section is based on the difference between the performance measure obtained from the LQN model and the corresponding estimate (as discussed in Section 3.6) from the PEPA model, according to the following definition of percentage relative error:

\[
\text{Error \%} = \left| \frac{\text{PEPA metric} - \text{LQN metric}}{\text{LQN metric}} \right| \times 100.
\]

This study considered all of the analysis techniques available in both formalisms, with emphasis on the issue of scalability, i.e., the resilience of the solution methods to increases in the size of the model under consideration. Here, scalability was studied empirically by estimating the incremental cost (i.e., runtime) of solving models which maintain the same topology but with increasingly large resource concurrency levels of some of its components. The performance metrics of interest were the steady-state average utilisations of the three (multi-) processors, denoted by \(U(P\text{Client})\), \(U(P\text{Server})\), and \(U(P\text{FileServer})\). The comparison of throughput measures showed a very similar trend, and it is not reported here due to lack of space.

The results were obtained with the PEPA Eclipse Plugin [21] and the Layered Queueing Network Solver software package [18]. For statistical significance, the execution times of all analyses presented in this section were averaged over five independent runs on an ordinary desktop machine.

4.1 Accuracy of the Translation

The exact form of analysis of PEPA models is the numerical solution of the underlying Markov chain, which was compared against simulation of the LQN using the method of batch means with automatic blocking and imposing a termination condition of 1% radius at 95% confidence intervals. Unless otherwise stated, these are the parameters used in the remainder of this paper for the simulation of all LQN models. Given the rapid growth of the state space of the Markov chain with increasing population sizes, the multiplicity of tasks and processors was kept low in this validation study. However, insight into the sensitivity of the accuracy was given by varying the execution demands in the model, which do not have an impact on the cardinality of the state space.

A crucial element in the PEPA model is \(\nu\), the only parameter which has no counterpart in the LQN model. Because of its semantics illustrated in the previous section, \(\nu\) is to be chosen such that the duration of the activities associated with this rate is negligible with respect to all other activities in the system. Table 2 shows the results of a sensitivity analysis conducted across an array of increasingly large values of \(\nu\). The slowest rate, i.e., \(1.2E04\), is equal to twenty times the fastest individual rate in the LQN model (i.e., one slice execution of external). The table reports the utilisations in terms of percentage relative differences with respect to the utilisations of the model with largest \(\nu\), i.e., \(1.2E08\). A tenfold increase to the value \(\nu\) corresponds to a decrease in the difference by the same factor, however for relatively small values of \(\nu\) the accuracy is very good, with discrepancies considerably less than one percent.

The level of precision obtained for \(\nu = 1.2E08\) was considered sufficient for the purposes of the present study. The results are presented in Table 3, which compares the processor utilisations for different execution demands and multiplicity of resources. Using the original concurrency levels, the accuracy improves by increasing the execution demand of cache, resulting in the entries of Server having similar overall execution demands. A noticeable reduction of the error can be noted when dem(cache) = 0.1, and adjusting the first phase of write brought only marginal improvements. Configuration B, featuring slightly larger population levels, presents more accurate results. Overall, there is good agreement between the two models, and despite the rather large numerical error in some instances, their qualitative behaviour is compatible.

4.2 Comparison of Simulation Approaches

The exponential growth of the state-space can be tackled by abandoning explicit enumeration in favour of stochastic simulation. PEPA has been equipped with a semantics which maps onto Gillespie’s simulation model [10], particularly suited for systems with large numbers of replicated agents [3]. The accuracy and scalability of this approach was compared against LQN simulation. Five instances of the model in Figure 1 were obtained by varying the multiplicity of tasks and processors, as listed in Table 4. The PEPA models were simulated using the method of independent replicas run over a sufficiently large time interval so as to reach equilibrium, using the same confidence-interval criterion used for the LQN model.

The results in Table 5 show the utilisation \(U(P\text{FileServer})\) and the runtimes obtained for each instance. This metric was arbitrarily chosen as a representative performance index, since all the other measures behaved very similarly, as can be observed by the results presented in Table 3. The

<table>
<thead>
<tr>
<th>(\nu)</th>
<th>(U(P\text{Client}))</th>
<th>(U(P\text{Server}))</th>
<th>(U(P\text{FileServer}))</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.2E08</td>
<td>0.093856</td>
<td>1.476351</td>
<td>0.680454</td>
</tr>
<tr>
<td>Relative differences</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.2E04</td>
<td>1.6846%</td>
<td>1.6846%</td>
<td>1.6846%</td>
</tr>
<tr>
<td>1.2E05</td>
<td>0.1708%</td>
<td>0.1714%</td>
<td>0.1708%</td>
</tr>
<tr>
<td>1.2E06</td>
<td>0.0169%</td>
<td>0.0170%</td>
<td>0.0170%</td>
</tr>
<tr>
<td>1.2E07</td>
<td>0.0015%</td>
<td>0.0015%</td>
<td>0.0015%</td>
</tr>
</tbody>
</table>

Table 2: Sensitivity of rate \(\nu\) in the PEPA model of Figure 1. First row: reference values. Other rows: relative differences with respect to first row.
Table 4: Model configurations of the LQN in Figure 1.

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Component</th>
<th>C1</th>
<th>C2</th>
<th>C3</th>
<th>C4</th>
<th>C5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Client</td>
<td></td>
<td>2</td>
<td>10</td>
<td>50</td>
<td>200</td>
<td>1000</td>
</tr>
<tr>
<td>Server</td>
<td></td>
<td>2</td>
<td>2</td>
<td>8</td>
<td>20</td>
<td>100</td>
</tr>
<tr>
<td>FileServer</td>
<td></td>
<td>2</td>
<td>2</td>
<td>8</td>
<td>20</td>
<td>50</td>
</tr>
<tr>
<td>Backup</td>
<td></td>
<td>2</td>
<td>2</td>
<td>8</td>
<td>20</td>
<td>30</td>
</tr>
<tr>
<td>PClient</td>
<td></td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>10</td>
<td>30</td>
</tr>
<tr>
<td>PServer</td>
<td></td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>10</td>
<td>30</td>
</tr>
<tr>
<td>PFileServer</td>
<td></td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>10</td>
<td>30</td>
</tr>
</tbody>
</table>

4.3 Comparison of Approximate Techniques

The main advantage in using approximate techniques is solution efficiency. This section discusses the MVA approach for LQNs and the fluid-flow approximation of PEPA based on ordinary differential equations [15]. Similarly to the previous section, the comparison considers the computational cost as well as the accuracy of these forms of analysis using the model configurations listed in Table 4. The differential equations were numerically integrated using an implementation of the adaptive step-size fifth-order Dormand-Prince algorithm [8], over an interval of five time units, which ensured equilibrium in all cases. For this analysis, the value of \( \nu \) was set to 1.2E05. The default parameters of the LQN analytical solver were not satisfactory for this study, instead Conway’s algorithm [5] was used and the solver option “stop-on-message-loss” was turned on to deal with the asynchronous requests at Server.

Table 6 shows the estimates of the utilisation of PFileServer, with percentage errors calculated with respect to the averages obtained by simulation of the LQN model in Table 5. In these instances, fluid-flow analysis is consistently more accurate than MVA. The error trend of the fluid-flow approximation of PEPA suggests that it behaves better as the population sizes in the system increase, reflecting several general results on the deterministic convergence of stochastic processes (e.g., [6]). Furthermore, the computational cost of fluid-flow analysis is low and largely independent from the population sizes. However, the numerical integration of the differential equation was found to be stiff with respect to \( \nu \), causing solution runtimes to grow proportionally with its actual value. For this reason, the value 1.2E05 was considered to be a better candidate than 1.2E08 in the trade-off between accuracy and solution efficiency.

In contrast to fluid-flow analysis, the execution runtimes for MVA were dependent upon the system size, although they were in general significantly faster than fluid-flow analysis (between about four and thirty times for configurations C1–C4 and executing with comparable runtime for configuration C5). According to other experiences published in the literature [9], models with such approximation errors as those reported here can be considered as being problematic with respect to the applicability of MVA, and in general one should expect more accurate results (i.e., within 5%). Nevertheless, these slightly large approximation errors in such particularly unfavourable instances are an adequate price to pay for the high efficiency of this solution technique.

4.4 Discussion

The numerical investigation presented in this paper suggests that the PEPA translation of LQN models offers...
Table 6: Comparison between MVA and fluid-flow analysis for the estimation of $U(P_{FileServer})$. Percentage relative errors calculated with respect to the simulation results in Table 5.

<table>
<thead>
<tr>
<th>Conf.</th>
<th>Metric (s)</th>
<th>Error</th>
<th>Metric (s)</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C1$</td>
<td>0.9142</td>
<td>17.5</td>
<td>17.56%</td>
<td>0.6177</td>
</tr>
<tr>
<td>$C2$</td>
<td>0.9157</td>
<td>17.7</td>
<td>10.34%</td>
<td>0.7123</td>
</tr>
<tr>
<td>$C3$</td>
<td>0.9150</td>
<td>16.8</td>
<td>0.76%</td>
<td>1.0945</td>
</tr>
<tr>
<td>$C4$</td>
<td>4.5771</td>
<td>20.9</td>
<td>0.71%</td>
<td>4.1717</td>
</tr>
<tr>
<td>$C5$</td>
<td>13.7303</td>
<td>17.4</td>
<td>0.71%</td>
<td>11.1700</td>
</tr>
</tbody>
</table>

As observed in Table 5, the rapid growth of the LQN simulation time with increasing concurrency levels indicates that PEPA stochastic simulation is preferred for the analysis of systems with many independent replicas, for which results are provided with very good accuracy. Conversely, LQN simulation is the method of choice when the multiplicities levels are relatively low, since the execution runtimes may be some orders of magnitude smaller.

More interesting is the comparison between MVA and fluid-flow analysis. Although both techniques are sufficiently accurate, fluid-flow analysis behaves remarkably well in the instances analysed in Section 4.3, especially in cases exhibiting components with an appreciable number of replicas. Fluid-flow analysis has very strong resilience to increases in the multiplicity levels, executing in similar lengths of time across the whole validation set. Again, for this reason this solution technique is more desirable than MVA for large-scale systems. In smaller models fluid-flow analysis appears to be less advantageous because of its higher computational cost. However, it should be noted that the execution time is a function of the integration interval, which was here set conservatively to five time units to ensure steady-state conditions in all cases. Thus, speed-up of fluid-flow analysis should be expected if the numerical integrator employed termination conditions based on the detection of equilibrium points for the solution.

Nevertheless, fluid-flow analysis in smaller models may be still preferred over MVA because it also provides transient measures of performance, extracted from the solution of the differential equation over a finite time interval. This information can be used to reason about different quantitative characteristics, such as warm-up periods (defined as the time interval necessary to reach equilibrium from some initial condition) and peak throughputs and utilisations. An example is shown in Figure 9, which plots the temporal evolution of the utilisation of the processors over the first two time units for the model configuration $C5$, clearly identifying $P_{Server}$ as the bottleneck of the system since almost all (i.e., 29.74) of the available processors are kept busy after a warm-up period of about 0.02 time units.

5. CONCLUSIONS

This paper presented an interpretation of LQNs as PEPA process algebra models. It supports a generous subset of the LQN model, including: synchronous and asynchronous request types, multiplicity of tasks and processors, two-phase activities, and execution graphs for the description of sequentiaility, conditional branching, and fork/join synchronisation. Ongoing work is concerned with extending this approach to other features not considered here, such as looping in execution graphs, synchronisation based on quorum consensus mechanisms, and forwarded replies (whereby the reply of one entry is delegated to some other entry in the network). The interpretation of the request count parameter corresponds to the deterministic semantics of the LQN model, i.e., the request is performed exactly the number of times shown in the request label. This is being extended to include requests with geometrical distributions. Finally, here all execution demands are assumed to be distributed exponentially, although the LQN model supports activities with arbitrary variance. This extension can be included in the present approach by using suitable phase-type distributions.

The validation conducted on a model which incorporates all of the supported features gave confidence on the soundness of the translation. Most important, it has shown that the solution methods of PEPA can be exploited in the LQN context to improve the efficiency of the analysis under the condition of models with large numbers of replicas, as well...
6. ACKNOWLEDGEMENT

The author thanks Stephen Gilmore for his comments on an earlier draft of this paper.

This work is supported by the EU FET-IST Global Computing 2 project SENSORIA (“Software Engineering for Service-Oriented Overlay Computers”) IST-3-016004-IP-09).

7. REFERENCES


APPENDIX

A. COMPLETE PEPA MODEL

Client (Reference Task)

\[ \text{Client}_1 \triangleq (\text{acquire}_{pc}, \nu). (\text{think}, 8, 0.01). \text{Client}_2 \]

\[ \text{Client}_2 \triangleq (\text{request}_{\text{think}}, \text{visit}, \nu). (\text{reply}_{\text{think}}, \text{visit}, \nu). (\text{acquire}_{pc}, \nu). (\text{think}, 8, 0.01). \text{Client}_3 \]

\[ \text{Client}_3 \triangleq (\text{request}_{\text{think}}, \text{visit}, \nu). (\text{reply}_{\text{think}}, \text{visit}, \nu). (\text{acquire}_{pc}, \nu). (\text{think}, 8, 0.01). \text{Client}_4 \]

\[ \text{Client}_4 \triangleq (\text{request}_{\text{think}}, \text{buy}, \nu). (\text{reply}_{\text{think}}, \text{buy}, \nu). (\text{acquire}_{pc}, \nu). (\text{think}, 8, 0.01). \text{Client}_5 \]

\[ \text{Client}_5 \triangleq (\text{request}_{\text{think}}, \text{notify}, \nu). (\text{acquire}_{pc}, \nu). (\text{think}, 8, 0.01). \text{Client}_6 \]

\[ \text{Client}_6 \triangleq (\text{request}_{\text{think}}, \text{notify}, \nu). (\text{reply}_{\text{think}}, \text{notify}, \nu). (\text{acquire}_{pc}, \nu). (\text{think}, 8, 0.01). \text{Client}_7 \]

\[ \text{Client}_7 \triangleq (\text{request}_{\text{think}}, \text{buy}, \nu). (\text{reply}_{\text{think}}, \text{buy}, \nu). (\text{acquire}_{pc}, \nu). (\text{think}, 8, 0.01). \text{Client}_8 \]

\[ \text{Client}_8 \triangleq (\text{request}_{\text{think}}, \text{read}, \nu). (\text{reply}_{\text{think}}, \text{read}, \nu). (\text{acquire}_{pc}, \nu). (\text{think}, 8, 0.01). \text{Client}_9 \]
Server

\[
\text{Server} \stackrel{def}{=} (\text{request}_{\text{think}}, \text{visit}, \nu). \text{Cache}_1
+ (\text{request}_{\text{think}}, \text{buy}, \nu). \text{Prepare}_1
+ (\text{request}_{\text{think}}, \text{notify}, \nu). \text{Notify}_1
+ (\text{request}_{\text{think}}, \text{save}, \nu). \text{Save}_1
\]

\[
\text{Cache}_1 \stackrel{def}{=} (\text{acquire}_{ps}, \nu).
\begin{align*}
&\left[ (\text{cache}, 0.95 \times 1/0.001). \text{Internal}_1 \\
&+ (\text{cache}, 0.05 \times 1/0.001). \text{External}_1 \right]
\end{align*}
\]

\[
\text{Internal}_1 \stackrel{def}{=} (\text{acquire}_{ps}, \nu). (\text{internal}, 1/0.001). \text{Internal}_2
\]

\[
\text{Internal}_2 \stackrel{def}{=} (\text{reply}_{\text{think}}, \text{visit}, \nu). \text{EndInternal}
\]

\[
\text{EndInternal} \stackrel{def}{=} \text{Server}
\]

\[
\text{Exernal}_1 \stackrel{def}{=} (\text{acquire}_{ps}, \nu). (\text{external}, 2/0.001). \text{External}_2
\]

\[
\text{Exernal}_2 \stackrel{def}{=} (\text{request}_{\text{external}}, \text{read}, \nu). (\text{reply}_{\text{external}}, \text{read}, \nu).
(\text{acquire}_{ps}, \nu). (\text{external}, 2/0.001). \text{External}_3
\]

\[
\text{Exernal}_3 \stackrel{def}{=} (\text{reply}_{\text{think}}, \text{visit}, \nu). \text{EndExternal}
\]

\[
\text{EndExternal} \stackrel{def}{=} \text{Server}
\]

\[
\text{Prepare}_1 \stackrel{def}{=} (\text{acquire}_{ps}, \nu). (\text{prepare}, 1/0.01). \text{ForkPrepare}
\]

\[
\text{ForkPrepare} \stackrel{def}{=} (\text{fork}_1, \nu). \text{EndPrepare}
\]

\[
\text{Display}_1 \stackrel{def}{=} (\text{acquire}_{ps}, \nu). (\text{display}, 1/0.001). \text{Display}_2
\]

\[
\text{Display}_2 \stackrel{def}{=} (\text{reply}_{\text{think}}, \text{buy}, \nu). \text{EndDisplay}
\]

\[
\text{EndDisplay} \stackrel{def}{=} \text{Server}
\]

\[
\text{Notify}_1 \stackrel{def}{=} (\text{acquire}_{ps}, \nu). (\text{notify}, 1/0.08). \text{EndNotify}
\]

\[
\text{EndNotify} \stackrel{def}{=} \text{Server}
\]

\[
\text{Save}_1 \stackrel{def}{=} (\text{acquire}_{ps}, \nu). (\text{save}, 2/0.02). \text{Save}_2
\]

\[
\text{Save}_2 \stackrel{def}{=} (\text{request}_{\text{save}}, \text{write}, \nu). (\text{reply}_{\text{save}}, \text{write}, \nu).
(\text{acquire}_{ps}, \nu). (\text{save}, 2/0.02). \text{Save}_3
\]

\[
\text{Save}_3 \stackrel{def}{=} (\text{reply}_{\text{think}}, \text{save}, \nu). \text{EndSave}
\]

\[
\text{EndSave} \stackrel{def}{=} \text{Server}
\]

Server's Secondary Flows

\[
\text{Pack}_1 \stackrel{def}{=} (\text{fork}_1, \nu). (\text{acquire}_{ps}, \nu). (\text{pack}, 1/0.03). \text{EndPack}
\]

\[
\text{EndPack} \stackrel{def}{=} (\text{join}_1, \nu). \text{Pack}_1
\]

\[
\text{Ship}_1 \stackrel{def}{=} (\text{fork}_1, \nu). (\text{acquire}_{ps}, \nu). (\text{ship}, 1/0.01). \text{EndShip}
\]

\[
\text{EndShip} \stackrel{def}{=} (\text{join}_1, \nu). \text{Ship}_1
\]

FileServer

\[
\text{FileServer} \stackrel{def}{=} (\text{request}_{\text{think}}, \text{read}, \nu). \text{Read}_1
+ (\text{request}_{\text{external}}, \text{read}, \nu). \text{Read}_2
+ (\text{request}_{\text{save}}, \text{write}, \nu). \text{Write}_1
\]

\[
\text{Read}_1 \stackrel{def}{=} (\text{acquire}_{ps}, \nu). (\text{read}, 1/0.001). \text{Read}_2
\]

\[
\text{Read}_2 \stackrel{def}{=} (\text{reply}_{\text{think}}, \text{read}, \nu). \text{EndRead}
+ (\text{reply}_{\text{external}}, \text{read}, \nu). \text{EndRead}
\]

\[
\text{EndRead} \stackrel{def}{=} \text{FileServer}
\]

\[
\text{Write}_1 \stackrel{def}{=} (\text{acquire}_{ps}, \nu). (\text{write}, 1/0.001). \text{Write}_2
\]

\[
\text{Write}_2 \stackrel{def}{=} (\text{reply}_{\text{save}}, \text{write}, \nu). \text{EndWrite}
\]

\[
\text{EndWrite} \stackrel{def}{=} \text{FileServer}
\]

\[
\text{Backup} \stackrel{def}{=} (\text{request}_{\text{write}}, \text{get}, \nu). \text{Get}_1
+ (\text{request}_{\text{write}}, \text{update}, \nu). \text{Update}_1
\]

\[
\text{Get}_1 \stackrel{def}{=} (\text{acquire}_{ps}, \nu). (\text{get}, 1/0.01). \text{Get}_2
\]

\[
\text{Get}_2 \stackrel{def}{=} (\text{reply}_{\text{write}}, \text{get}, \nu). \text{EndGet}
\]

\[
\text{EndGet} \stackrel{def}{=} \text{Backup}
\]

\[
\text{Update}_1 \stackrel{def}{=} (\text{acquire}_{ps}, \nu). (\text{update}, 1/0.01). \text{Update}_2
\]

\[
\text{Update}_2 \stackrel{def}{=} (\text{reply}_{\text{write}}, \text{update}, \nu). \text{EndUpdate}
\]

\[
\text{EndUpdate} \stackrel{def}{=} \text{Backup}
\]

PClient

\[
\text{PClient} \stackrel{def}{=} (\text{acquire}_{ps}, \nu). \text{PClient}'
\]

\[
\text{PClient}' \stackrel{def}{=} (\text{think}, 8/0.01). \text{PClient}
\]

PServer

\[
\text{PServer} \stackrel{def}{=} (\text{cache}, 1/0.001). \text{PServer'}
\]

\[
\text{PServer}' \stackrel{def}{=} (\text{internal}, 1/0.001). \text{PServer}
\]

\[
\text{PServer} \stackrel{def}{=} (\text{prepare}, 1/0.01). \text{PServer'}
\]

\[
\text{PServer} \stackrel{def}{=} (\text{pack}, 1/0.03). \text{PServer'}
\]

\[
\text{PServer} \stackrel{def}{=} (\text{display}, 1/0.001). \text{PServer'}
\]

PFileServer: (cfr. Figure 3)

\[
\text{Complete Layered Queuing Network}
\]

\[
\begin{align*}
\text{Client}[2] \rightarrow \text{M}_1 \rightarrow \{ & \text{Server}[2] \rightarrow \text{L}_1 \rightarrow \text{Pack}_1 [2] \rightarrow \text{M}_2 \rightarrow \text{Ship}_1 [2] \rightarrow \text{M}_3 \rightarrow \text{Backup}[1] \\
& \text{M}_4 \rightarrow \{ \text{PClient}[2] \rightarrow \text{PServer}[2] \rightarrow \text{PFileServer}[2] \} \}
\end{align*}
\]

\[
\text{M}_1 = \{ \text{request}_{\text{think}}, \text{visit}, \text{reply}_{\text{think}}, \text{visit}, \text{request}_{\text{think}}, \text{buy}, \text{request}_{\text{think}}, \text{notify}, \text{request}_{\text{think}}, \text{save}, \text{reply}_{\text{think}}, \text{save} \}
\]

\[
\text{L}_1 = \text{L}_2 = \{ \text{fork}_1, \text{join}_1 \}
\]

\[
\text{M}_2 = \{ \text{request}_{\text{think}}, \text{read}, \text{reply}_{\text{think}}, \text{read}, \text{request}_{\text{external}}, \text{read} \}
\]

\[
\text{M}_3 = \{ \text{request}_{\text{write}}, \text{get}, \text{reply}_{\text{write}}, \text{get}, \text{request}_{\text{write}}, \text{update}, \text{reply}_{\text{write}}, \text{update} \}
\]

\[
\text{M}_4 = \{ \text{acquire}_{ps}, \text{think}, \text{acquire}_{ps}, \text{cache}, \text{internal}, \text{external}, \text{prepare}, \text{pack}, \text{ship}, \text{display}, \text{notify}, \text{display}, \text{acquire}_{ps}, \text{read}, \text{write}, \text{get}, \text{update} \}
\]