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 advatanges 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 [23]. 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-

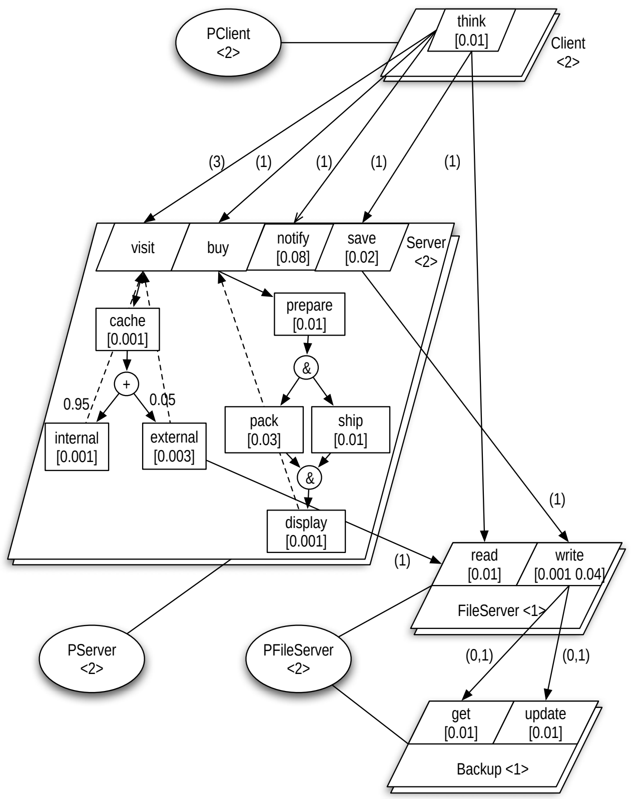

Figure 1: LQN model of a distributed application.

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-*

Table 1: Summary of notation.

Symbol	*Meaning*	*Meta-variables*
$\mathcal{A}$	Set of LQN activities	a
$\mathcal{E}$	Set of LQN entries	e
$\mathcal{K}$	Synchronicity type. $\mathcal{K} = \{\text{sync}, \text{async}\}$	k
$\mathcal{P}$	Set of LQN processors	p
$\mathcal{T}$	Set of LQN tasks	t
$\text{act}(p) : \mathcal{P} \to 2^{\mathcal{A}}$	Set of activities executed on p	
$\text{act}^{-1}(a) : \mathcal{A} \to \mathcal{P}$	Process on which a is executed	
$\text{dem}(a) : \mathcal{A} \to \mathbb{R}_{\geq 0}$	Total execution demand of activity a	
$\text{ent}(t) : \mathcal{T} \to 2^{\mathcal{E}}$	Set of entries in task t	
$\text{mpr}(p) : \mathcal{P} \to \mathbb{N}$	Multiplicity of processor p	
$\text{mtk}(t) : \mathcal{T} \to \mathbb{N}$	Multiplicity of task t	
$\text{rep}(a) : \mathcal{A} \to \mathcal{E}$	The entry to which activity a replies (may be empty)	
$\text{req}(a) : \mathcal{A} \to 2^{\mathcal{E} \times \mathbb{N} \times \mathcal{K}}$	Set of requests made by activity a	(e, n, k)
$N(a) = \sum_{(e,n,k) \in \text{req}(a)} n$	Total number of requests made during activity a	

tions) or in synchronisation with other sequential components of the system (*shared actions*). The language supports the following operators:

Prefix $(\alpha, r).P$ constitutes the atomic unit of computation of a PEPA model. It is a sequential component which may perform an action of type α, 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 $(\alpha, r).P + (\beta, s).Q$ is said to *enable* actions α and β, 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*:

$$\sum \{(\alpha, r).P \,|\, \mathbb{P}(\alpha, r, P)\},$$

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

$$\sum \{(\alpha, i \cdot r).P_i \,|\, 1 \leq i \leq 2)\} \stackrel{def}{=} (\alpha, r).P_1 + (\alpha, 2r).P_2$$

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

Cooperation $P \underset{L}{\bowtie} Q$ 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 $(\alpha, r).(\beta, s).P \underset{\{\alpha\}}{\bowtie} (\alpha, t).(\gamma, u).Q$ is a cooperation between two sequential components which may perform a shared activity of type α, subsequently behaving as $(\beta, s).P \underset{\{\alpha\}}{\bowtie} (\gamma, u).Q$. Then actions β and γ are carried autonomously. By contrast, in the cooperation $(\alpha, r).P \underset{\{\alpha\}}{\bowtie} (\beta, s).Q$ the process $(\alpha, r).P$ does not progress because α 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 $*$. 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 τ. 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 $\underbrace{T \underset{\emptyset}{\bowtie} T \underset{\emptyset}{\bowtie} \cdots \underset{\emptyset}{\bowtie} T}_{N}$, 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.

$$\begin{aligned}
\mathit{Proc}_p &\overset{\mathit{def}}{=} (\mathit{acquire}_p, \nu).\mathit{Exec}_p \\
\mathit{Exec}_p &\overset{\mathit{def}}{=} \sum_{a \in \mathrm{act}(p) : \mathrm{dem}(a) > 0} \left(a, (N(a)+1) \cdot \mathrm{dem}(a)^{-1}\right).\mathit{Proc}_p
\end{aligned}$$

Figure 2: Translation of an LQN Processor.

$$\begin{aligned}
\mathit{PFileServer}' &\overset{\mathit{def}}{=} (\mathit{acquire}_{\mathit{pfs}}, \nu).\mathit{PFileServer}'' \\
\mathit{PFileServer}'' &\overset{\mathit{def}}{=} (\mathit{read}, 1/0.01).\mathit{PFileServer}' \\
&\quad + (\mathit{write}_{\mathit{1}}, 1/0.001).\mathit{PFileServer}' \\
&\quad + (\mathit{write}_{\mathit{2}}, 3/0.04).\mathit{PFileServer}' \\
&\quad + (\mathit{get}, 1/0.01).\mathit{PFileServer}' \\
&\quad + (\mathit{update}, 1/0.01).\mathit{PFileServer}'
\end{aligned}$$

Figure 3: Translation of *PFileServer*.

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 Proc_p models an activity which grants exclusive access to the processor. The rate ν 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 Exec_p enables all the actions corresponding to the activities which are executed on p, by means of the choice operator. Each activity phase is mapped onto a distinct action type in PEPA 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 Exec_p in Figure 2):

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

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 *PFileServer* is shown in Figure 3.

3.2 Activity and Request

An LQN activity subsumes a sequence of PEPA 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 PEPA action type for the request has the form $\mathit{request}_{a,e}$, 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 $\mathit{reply}_{a,e}$. An asynchronous call is represented with a single prefix of type $\mathit{request}_{a,e}$.

The PEPA process corresponding to the LQN activity interposes executions of slices of a between requests. The rates for requests and replies are here set to ν, 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 PEPA descriptions will be useful for the translation of an activity:

$$\mathit{Acq}_a \equiv \begin{cases} \left(\mathit{acquire}_{\mathrm{act}^{-1}(a)}, \nu\right).(a, s(a)) & \text{if } s(a) > 0 \\ \epsilon & \text{if } s(a) = 0 \end{cases}$$

$$\begin{aligned}
\mathit{Sync}_{a,e} &\equiv (\mathit{request}_{a,e}, \nu).(\mathit{reply}_{a,e}, \nu).\mathit{Acq}_a \\
\mathit{Async}_{a,e} &\equiv (\mathit{request}_{a,e}, \nu).\mathit{Acq}_a
\end{aligned}$$

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

$$\mathit{Act}^a_{\mathit{1}} \overset{\mathit{def}}{=} \mathit{Acq}_a.\mathit{Act}^a_{\mathit{2}}$$

$$\mathit{Act}^a_{i+\mathit{1}} \overset{\mathit{def}}{=} \begin{cases} \underbrace{\mathit{Sync}_{a,e_i}.\cdots.\mathit{Sync}_{a,e_i}}_{n_i}.\mathit{Act}^a_{i+\mathit{2}} & \text{if } k_i = \text{sync} \\ \underbrace{\mathit{Async}_{a,e_i}.\cdots.\mathit{Async}_{a,e_i}}_{n_i}.\mathit{Act}^a_{i+\mathit{2}} & \text{if } k_i = \text{async} \end{cases}$$

$$(e_i, n_i, k_i) \in \mathrm{req}(a), \text{for all } 1 \le i < |\mathrm{req}(a)|$$

Case $\mathrm{rep}(a) = \emptyset$:

$$\mathit{Act}^a_{|\mathrm{req}(a)|+\mathit{2}} \overset{\mathit{def}}{=} \mathit{End}_a$$

Case $\mathrm{rep}(a) \neq \emptyset$:

$$\mathit{Act}^a_{|\mathrm{req}(a)|+\mathit{2}} \overset{\mathit{def}}{=} \sum \Big\{ (\mathit{reply}_{a',\mathrm{rep}(a)}, \nu).\mathit{End}_a \;\Big|\; \exists (e,n,k) \in \mathrm{req}(a') : e = \mathrm{rep}(a), \forall a' \in \mathcal{A} \Big\}$$

Figure 4: Translation of an LQN Activity.

The translation of an LQN activity is shown in Figure 4. The first process definition $\mathit{Act}^a_{\mathit{1}}$ 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 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 *write1* and *write2*. Activity *write1* does not make requests to lower-level server but it replies to requests to entry *write* made by *get*. Activity *write2* 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 PEPA 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 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 computa-

First phase

$$Write'_1 \overset{def}{=} (acquire_{pfs}, \nu).(write_1, 1/0.001).Write'_2$$

$$Write'_2 \overset{def}{=} (reply_{save,write}, \nu).EndWrite'$$

Second phase

$$Write''_1 \overset{def}{=} (acquire_{pfs}, \nu).(write_2, 3/0.04).Write''_2$$

$$Write''_2 \overset{def}{=} (request_{write_2,get}, \nu).(reply_{write_2,get}, \nu).(acquire_{pfs}, \nu).(write_2, 3/0.04).Write''_3$$

$$Write''_3 \overset{def}{=} (request_{write_2,update}, \nu).(reply_{write_2,update}, \nu).(acquire_{pfs}, \nu).(write_2, 3/0.04).EndWrite''$$

Figure 5: Translation of activity *write*.

tion in an execution graph. However, while an action node is translated into a single PEPA 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 PEPA constants which define the initial and the final state (i.e., Act^a_1 and End_a 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 *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 $Main \underset{L}{\bowtie} S$, where S is an arbitrary PEPA process consisting of the sequential components which model the remaining control flows, called *secondary flows*.

Precedence

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 PEPA 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 *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 $EndWrite' \overset{def}{=} Write''_1$ (recall that $EndWrite'$ was left intentionally unspecified for this purpose).

Probabilistic Branching

The translation of probabilistic branching (called *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, *cache* is translated into a sequential component in the simple form

$$Cache_1 \overset{def}{=} (acquire_{ps}, \nu).(cache, 1/0.001).EndCache.$$

Being the predecessor of a branching operator, its last activity *cache* is replaced with a PEPA choice as follows:

$$(cache, 1/0.001).EndCache \rightarrow (cache, 0.95 \times 1/0.001).Internal_1 + (cache, 0.05 \times 1/0.001).External_1$$

This component is capable of performing the activity *cache* at the original rate 1/0.001 (obtained as the sum of the two alternative behaviours), but with probability 0.95 and 0.05 it then behaves as one of its successors, i.e., $Internal_1$ and $External_1$, respectively.

Alternative behaviours may merge back into one (*Or-join* operator). This is translated in PEPA by letting all of the final states of the merging elements coincide with the initial state of the merged behaviour. Or-join nodes are not used in the running example.

Fork/Join Synchronisation

The presence of a fork/join synchronisation mechanism implies that an entry has explicit concurrent behaviour. This is captured in PEPA by assigning a sequential component to each distinct concurrent control flow. Such flows perform the activities autonomously and synchronise over action types corresponding to fork and join nodes in the execution graph. A basic activity is uniquely assigned to one flow and the algorithm keeps track of the initial state of all flows. This is necessary to define the constituting sequential components in a cyclic manner. The initial activity of an entry's execution graph is said to start the *main control flow* of the entry. All subsequent activities are executed within the same control flow as is the case for the entry *visit*. Conversely, the entry *buy* has three control flows. In addition to the main one started by *prepare*, two further are spawned by the fork operator. Their initial states are $Prepare_1$, $Pack_1$, and $Ship_1$, respectively defined as follows:

$$Prepare_1 \overset{def}{=} (acquire_{ps}, \nu).(prepare, 1/0.01).EndPrepare$$

$$Pack_1 \overset{def}{=} (acquire_{ps}, \nu).(pack, 1/0.03).EndPack$$

$$Ship_1 \overset{def}{=} (acquire_{ps}, \nu).(ship, 1/0.01).EndShip$$

As with probabilistic branching, the translation of a fork operator takes as input the set of activities enabled by final 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 *fork* activity, carried out at rate ν, indicating a negligible rate of spawning new processes. Each state in the latter set is instead modified so as to have *fork* as the first enabled activity. For instance, the PEPA component corresponding to the basic activity *prepare* is modified to become

$$Prepare_1 \overset{def}{=} (acquire_{ps}, \nu).(prepare, 1/0.01).ForkPrepare$$

$$ForkPrepare \overset{def}{=} (fork_1, \nu).EndPrepare$$

where the subscript in the action $fork_1$ is used to uniquely assign a type to each fork node in the execution graph, for instance by mapping them into integers. Similarly, $Pack_1$ and $Ship_1$ are prefixed with the $fork_1$, i.e.,

$$Pack_1 \overset{def}{=} (fork_1, \nu).(acquire_{ps}, \nu).(pack, 1/0.03).EndPack$$

$$Ship_1 \overset{def}{=} (fork_1, \nu).(acquire_{ps}, \nu).(ship, 1/0.01).EndShip$$

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 ν) and subsequently cycle back to the flows' initial states. In the example,

$$\mathit{EndPack} \overset{def}{=} (\mathit{join_1}, \nu).\mathit{Pack_1}$$
$$\mathit{EndShip} \overset{def}{=} (\mathit{join_1}, \nu).\mathit{Ship_1}$$

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

$$\mathit{Display_1} \overset{def}{=} (\mathit{acquire_{ps}}, \nu).(\mathit{display}, 1/0.001).\mathit{EndDisplay}$$

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

$$\mathit{EndPrepare} \overset{def}{=} (\mathit{join_1}, \nu).\mathit{Display_1}$$

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:

$$\mathit{Prepare_1} \underset{L}{\bowtie} \big(\mathit{Pack_1}[2] \underset{L}{\bowtie} \mathit{Ship_1}[2]\big), \quad L = \{\mathit{fork_1}, \mathit{join_1}\}$$

The overall behaviour of *visit*, consisting of a single flow of control is simply represented by the flow's initial state *Cache₁*. In the remainder of this paper, the overall model of an execution graph will be denoted by the component $\mathit{Main_e} \underset{L_e}{\bowtie} \mathit{Sec_e}$, where $\mathit{Main_e}$ is the behaviour of the main flow, without information on its multiplicity, and $\mathit{Sec_e}$ comprises all the secondary flows, with proper multiplicites. 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 $\mathit{EndWrite''}$ in *write*. For an entry e such a constant will be denoted by $\mathit{Last_e}$.

3.4 Task

A reference task, here denoted by t^*, has the same behaviour as its unique entry, denoted by e^*. 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 *Read₁* (the actual behaviour of the entry is independent from the originator of the request).

Reference Task

$$\mathit{Task_{t^*}} \overset{def}{=} \mathit{Main_{e^*}}$$
$$\mathit{Last_{e^*}} \overset{def}{=} \mathit{Task_{t^*}}$$

Non-Reference Task

$$\mathit{Task_t} \overset{def}{=} \sum \Big\{ (\mathit{request_{a,e}}, \nu).\mathit{Main_e} \mid \exists (e,n,k) \in \mathrm{req}(a) : e \in \mathrm{ent}(t), \forall a \in \mathcal{A} \Big\}$$
$$\mathit{Last_e} \overset{def}{=} \mathit{Task_t}, \quad \text{for each } e \in \mathrm{ent}(t)$$

Figure 6: Translation of an LQN Task.

$$\begin{aligned}
\mathit{FileServer} &\overset{def}{=} (\mathit{request_{external,read}}, \nu).\mathit{Read_1} \\
&\quad + (\mathit{request_{think,read}}, \nu).\mathit{Read_1} \\
&\quad + (\mathit{request_{save,write_1}}, \nu).\mathit{Write'_1} \\
\mathit{Read_1} &\overset{def}{=} (\mathit{acquire_{pfs}}, \nu).(\mathit{read}, 1/0.01).\mathit{EndRead} \\
\mathit{EndRead} &\overset{def}{=} \mathit{FileServer} \\
\mathit{Write'_1} &\overset{def}{=} \ldots \\
\ldots &\quad \text{(cfr. Figure 5)} \\
\mathit{EndWrite''} &\overset{def}{=} \mathit{FileServer}
\end{aligned}$$

Figure 7: Translation of task *FileServer*.

3.5 Network

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

$$\mathit{Comp_t} \overset{def}{=} \mathit{Task_t}[\mathrm{mtk}(t)] \underset{\widehat{L}}{\bowtie} \Big(\mathit{Sec_{e_1}} \underset{\emptyset}{\bowtie} \mathit{Sec_{e_2}} \underset{\emptyset}{\bowtie} \cdots \underset{\emptyset}{\bowtie} \mathit{Sec_{e_{|\mathrm{ent}(t)|}}}\Big)$$

where $\widehat{L} = \bigcup_{i=1}^{|\mathrm{ent}(t)|} L_{e_i}$, for all $t \in \mathcal{T}$

$$\mathit{LQN} \overset{def}{=} \Big(\mathit{Comp_{t_1}} \underset{\widehat{M}}{\bowtie} \mathit{Comp_{t_2}} \cdots \underset{\widehat{M}}{\bowtie} \mathit{Comp_{t_{|\mathcal{T}|}}}\Big) \underset{*}{\bowtie} \Big(\mathit{Proc_{p_1}}[\mathrm{mpr}(p_1)] \underset{\emptyset}{\bowtie} \mathit{Proc_{p_2}}[\mathrm{mpr}(p_2)] \underset{\emptyset}{\bowtie} \cdots \underset{\emptyset}{\bowtie} \mathit{Proc_{|\mathcal{P}|}}[\mathrm{mpr}(p_{|\mathcal{P}|})]\Big),$$

where, $\widehat{M} = * - \bigcup_{p \in \mathcal{P}} \{\mathit{acquire_p}\}$.

Figure 8: Translation of an LQN.

in Figure 8. The definition $Comp_t$ describes the overall behaviour of a multithreaded server with multiple entries. The task behaviour $Task_t$ (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, $Task_t$ 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 $Comp_t$ are combined together using cooperation sets which can be denoted by the same expression $\widehat{M}$. However, notice that the actual instantiations are all different because of the dependence of the set $*$ 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 $acquire_p$ action types are not contained in the sets $\widehat{M}$. Any pair of components of type $Comp_t$ 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 $\widehat{M}$ 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 $Comp_t$, i.e.,

$$Comp_{t_1} \underset{\widehat{M}}{\bowtie} Comp_{t_2} \underset{\widehat{M}}{\bowtie} Comp_{t_3},$$

component $Comp_{t_1}$ may enable request/reply actions with subscripts (a', e'), (a'', e''), $\ldots$, where $e', e'', \ldots \in \mathrm{ent}(t_1)$ 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 $Comp_{t_2}$ and $Comp_{t_3}$ can perform the same basic activity a, which is a contradiction. Then, assuming that the property holds for a cooperation among $n > 3$ components

$$Comp_{t_1} \underset{\widehat{M}}{\bowtie} Comp_{t_2} \underset{\widehat{M}}{\bowtie} Comp_{t_3} \underset{\widehat{M}}{\bowtie} \cdots \underset{\widehat{M}}{\bowtie} Comp_{t_n},$$

in order to prove that it holds for $n + 1$ components

$$Comp_{t_1} \underset{\widehat{M}}{\bowtie} Comp_{t_2} \underset{\widehat{M}}{\bowtie} Comp_{t_3} \underset{\widehat{M}}{\bowtie} \cdots \underset{\widehat{M}}{\bowtie} Comp_{t_n} \underset{\widehat{M}}{\bowtie} Comp_{t_{n+1}},$$

it suffices to prove that the cooperation set $\widehat{M}$ in position $\cdots Comp_{t_i} \underset{\widehat{M}}{\bowtie} Comp_{t_i+1} \cdots$ is disjoint from the cooperation set $\cdots Comp_{t_n} \underset{\widehat{M}}{\bowtie} Comp_{t_{n+1}}$, for all $1 \le i \le n-1$. Suppose that for some i $Comp_{t_i} \underset{\widehat{M}}{\bowtie} Comp_{t_i+1}$ has some action in common with the set in $Comp_{t_n} \underset{\widehat{M}}{\bowtie} Comp_{t_{n+1}}$. This implies that the action must be a request/reply action with subscript (a, e), $e \in \mathrm{ent}(t_{n+1})$, because it belongs to the set $Comp_{t_n} \underset{\widehat{M}}{\bowtie} Comp_{t_{n+1}}$, and that $e \in \mathrm{ent}(t_{i+1})$, 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 $Comp_t$ synchronise 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 $Exec_p$ (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 $Exec_p$ (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 *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 $(write_1, 1/0.001).Write'_2$, whereas the utilisation during the second phase is the sum of the expectation of the following three sequential components: $(write_2, 3/0.04).Write''_2$, $(write_2, 3/0.04).Write''_3$, 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

Table 2: Sensitivity of rate ν in the PEPA model of Figure 1. First row: reference values. Other rows: relative differences with respect to first row.

ν	$U(PClient)$	$U(PServer)$	$U(PFileServer)$
Reference values			
1.2E08	**0.093856**	**1.476351**	**0.680454**
Relative differences			
1.2E04	1.6846%	1.6846%	1.6848%
1.2E05	0.1708%	0.1714%	0.1708%
1.2E06	0.0169%	0.0170%	0.0170%
1.2E07	0.0015%	0.0015%	0.0015%

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 $write_1$ and $write_2$, 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 $U(PClient)$, $U(PServer)$, and $U(PFileServer)$. 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 Plug-in* [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 ν, the only parameter which has no counterpart in the LQN model. Because of its semantics illustrated in the previous section, ν 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 ν. 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 ν, i.e., 1.2E08. A tenfold increase to the value ν corresponds to a decrease in the difference by the same factor, however for relatively small values of ν the accuracy is very good, with discrepancies considerably less than one percent.

The level of precision obtained for $\nu = 1.2\text{E}08$ 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 $U(PFileServer)$ 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 agreement improves with larger populations, giving excellent results as the system under study has tens or hundreds of clients and many server threads and processing resources.

Table 3: Accuracy of the translation of the LQN in Figure 1 (solution of the Markov chain in PEPA vs. simulation of the LQN). Concurrency configurations: (A) original model; (B) all multiplicities of tasks and processors set to two. The execution demands not shown in the table are set as in the original model.

Concurrency	*Demands*		U(*PClient*)			U(*PServer*)			U(*PFileServer*)		
configuration	dem(cache)	dem(write$_1$)	PEPA	LQN	Error	PEPA	LQN	Error	PEPA	LQN	Error
A	0.0001	0.0010	0.0950	0.0843	12.67%	1.4681	1.3024	12.72%	0.6888	0.6108	12.78%
A	0.0010	0.0010	0.0939	0.0825	12.44%	1.4763	1.3136	12.39%	0.6804	0.6052	12.42%
A	0.0100	0.0010	0.0844	0.0762	10.80%	1.5561	1.4034	10.88%	0.6121	0.5497	11.35%
A	0.1000	0.0010	0.0406	0.0385	5.57%	1.8465	1.7493	5.56%	0.2947	0.2786	5.79%
A	0.1000	0.0600	0.0349	0.0331	5.20%	1.5839	1.5045	5.28%	0.4585	0.4355	5.28%
B	0.0010	0.0010	0.1172	0.1075	9.03%	1.8440	1.6902	9.10%	0.8499	0.7800	8.96%
B	0.1000	0.0600	0.0380	0.0371	2.53%	1.7267	1.6840	2.54%	0.4998	0.4887	2.28%

Table 4: Model configurations of the LQN in Figure 1.

Configuration / *Component*	*C1*	*C2*	*C3*	*C4*	*C5*
Client	2	10	50	200	1000
Server	2	2	8	20	100
FileServer	2	2	8	20	50
Backup	2	2	8	20	30
PClient	2	2	2	10	30
PServer	2	2	2	10	30
PFileServer	2	2	2	10	30

Table 5: Simulation results for U(*PFileServer*).

Conf.	*PEPA*		*LQN*		*Error*
	Metric	(hh:mm:ss)	Metric	(hh:mm:ss)	
C1	0.8494	00:44:41	0.7792	00:00:41	9.05%
C2	0.8660	00:52:50	0.8307	00:01:36	2.34%
C3	0.9314	01:14:25	0.9208	00:08:55	1.14%
C4	4.6002	02:32:19	4.6101	00:39:07	0.21%
C5	13.8640	03:42:31	13.8280	04:47:12	0.26%

Furthermore, the stochastic simulation algorithm for PEPA is more scalable with respect to increases in the population levels of the model's components. For instance, the largest model was about five times as costly as the smallest one (whose population levels are about two orders of magnitude smaller), as opposed to a corresponding increase by a factor of over 400 in the runtime of the LQN simulation. However, in absolute terms LQN simulation was much faster than PEPA simulation in the first four cases, with runtimes of the same magnitude only for configuration *C5*.

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 fitfh-order Dormand-Prince algorithm [8], over an interval of five time units, which ensured equilibrium in all cases. For this analysis, the value of ν 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 ν, 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 complementary rather than competing analysis techniques for the performance evaluation of software systems. The original semantics of PEPA permits explicit enumeration of the

Table 6: Comparison between MVA and fluid-flow analysis for the estimation of $U(\textbf{\textit{PFileServer}})$. Percentage relative errors calculated with respect to the simulation results in Table 5.

Conf.	*PEPA*			*LQN*		
	Metric	(s)	Error	Metric	(s)	Error
C1	0.9142	17.5	17.50%	0.6177	0.4	20.59%
C2	0.9157	17.7	10.34%	0.7123	0.7	14.17%
C3	0.9150	16.8	0.76%	1.0895	1.1	18.16%
C4	4.5771	20.9	0.71%	4.1717	6.5	9.51%
C5	13.7303	17.4	0.71%	11.1700	22.5	19.22%

complete state space of the model, enabling forms of analysis, e.g., model-checking, which do not require the solution of a performance model, but nevertheless give insight into the qualitative behaviour of the system. In relatively small models for which the numerical solution of the underlying Markov chain is feasible, other indices of performance are possible beyond those considered in the LQN model. For instance, the technology of *stochastic probes* for PEPA supports passage-time analysis in which complex passages over the Markov chain can be described using a rich language based on regular expressions over the model's process-algebraic terms [1].

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 behaved 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 mutiplicity levels, executing in similar lengths of time across the whole validation set. Again, for this reason this solution techique 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 *PServer* 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.

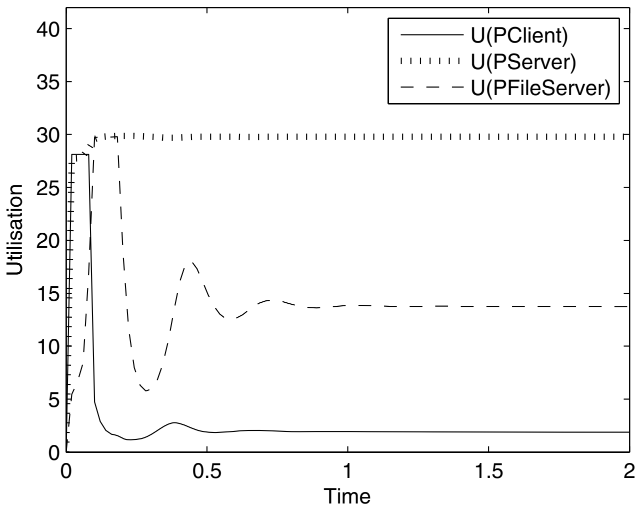

Figure 9: Temporal evolution of the utilisation of the processors of configuration *C5* over the first two time units.

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 sequentiality, 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 as to enrich the analysis with information on the transient behaviour of the system under study.

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

[1] A. Argent-Katwala, J. Bradley, and N. Dingle. Expressing performance requirements using regular expressions to specify stochastic probes over process algebra models. In *Proceedings of the Fourth International Workshop on Software and Performance*, pages 49–58, Redwood Shores, California, USA, Jan. 2004. ACM Press.

[2] M. Bernardo and R. Gorrieri. A tutorial on EMPA: A theory of concurrent processes with nondeterminism, priorities, probabilities and time. *Theor. Comput. Sci.*, 202(1-2):1–54, 1998.

[3] J. T. Bradley and S. T. Gilmore. Stochastic simulation methods applied to a secure electronic voting model. *Electr. Notes Theor. Comput. Sci.*, 151(3):5–25, 2006.

[4] K. M. Chandy and D. Neuse. Linearizer: A heuristic algorithm for queueing network models of computing systems. *Commun. ACM*, 25(2):126–134, 1982.

[5] A. Conway. Fast approximate solution of queueing networks with multi-server chain-dependent fcfs queues. In R. Puigjaner and D. Potier, editors, *Modeling Techniques and Tools for Computer Performance Evaluation*, pages 385–396, New York, 1989. Plenum.

[6] R. Darling and J. Norris. Differential equation approximations for Markov chains. *Probability Surveys*, 5:37–79, 2008.

[7] E. de Souza e Silva, S. S. Lavenberg, and R. R. Muntz. A clustering approximation technique for queueing network models with a large number of chains. *IEEE Trans. Computers*, 35(5):419–430, 1986.

[8] J. Dormand and P. Prince. A family of embedded Runge-Kutta formulae. *Journal of Computational and Applied Mathematics*, 6(1):19–26, March 1980.

[9] G. Franks, T. Omari, C. M. Woodside, O. Das, and S. Derisavi. Enhanced modeling and solution of layered queueing networks. *IEEE Trans. Software Eng.*, 35(2):148–161, 2009.

[10] D. Gillespie. Exact stochastic simulation of coupled chemical reactions. *Journal of Physical Chemistry*, 81(25):2340–2361, December 1977.

[11] S. Gilmore, J. Hillston, and M. Ribaudo. An efficient algorithm for aggregating PEPA models. *IEEE Transactions on Software Engineering*, 27(5):449–464, May 2001.

[12] N. Götz, U. Herzog, and M. Rettelbach. TIPP—a language for timed processes and performance evaluation. Technical Report 4/92, IMMD7, University of Erlangen-Nürnberg, Germany, Nov. 1992.

[13] U. Herzog and J. A. Rolia. Performance validation tools for software/hardware systems. *Perform. Eval.*, 45(2-3):125–146, 2001.

[14] J. Hillston. *A Compositional Approach to Performance Modelling*. Cambridge University Press, 1996.

[15] J. Hillston. Fluid flow approximation of PEPA models. In *Proceedings of the Second International Conference on the Quantitative Evaluation of Systems*, pages 33–43, Torino, Italy, Sept. 2005. IEEE Computer Society Press.

[16] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

[17] R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag, 1980.

[18] Real-Time and Distributed Systems group, Department of Systems and Computer Engineering, University of Carleton. LQNS software package. `http://www.sce.carleton.ca/rads/lqns`.

[19] M. Reiser and S. S. Lavenberg. Mean-value analysis of closed multichain queuing networks. *J. ACM*, 27(2):313–322, 1980.

[20] J. A. Rolia and K. C. Sevcik. The method of layers. *IEEE Trans. Software Eng.*, 21(8):689–700, 1995.

[21] M. Tribastone, A. Duguid, and S. Gilmore. The PEPA Eclipse Plug-in. *Performance Evaluation Review*, 36(4):28–33, March 2009.

[22] M. Tribastone and S. Gilmore. Automatic extraction of PEPA performance models from UML activity diagrams annotated with the MARTE profile. In A. Avritzer, E. J. Weyuker, and C. M. Woodside, editors, *Proceedings of the 7th International Workshop on Software and Performance, WOSP*, pages 67–78, Princeton NJ, USA, 2008. ACM.

[23] C. M. Woodside. Throughput calculation for basic stochastic rendezvous networks. *Perform. Eval.*, 9(2):143–160, 1989.

APPENDIX

A. COMPLETE PEPA MODEL

Client (Reference Task)

$$\begin{aligned}
Client_1 &\overset{def}{=} (acquire_{pc}, \nu).(think, 8/0.01).Client_2\\
Client_2 &\overset{def}{=} (request_{think,visit}, \nu).(reply_{think,visit}, \nu).\\
&\qquad (acquire_{pc}, \nu).(think, 8/0.01).Client_3\\
Client_3 &\overset{def}{=} (request_{think,visit}, \nu).(reply_{think,visit}, \nu).\\
&\qquad (acquire_{pc}, \nu).(think, 8/0.01).Client_4\\
Client_4 &\overset{def}{=} (request_{think,visit}, \nu).(reply_{think,visit}, \nu).\\
&\qquad (acquire_{pc}, \nu).(think, 8/0.01).Client_5\\
Client_5 &\overset{def}{=} (request_{think,buy}, \nu).(reply_{think,buy}, \nu).\\
&\qquad (acquire_{pc}, \nu).(think, 8/0.01).Client_6\\
Client_6 &\overset{def}{=} (request_{think,notify}, \nu).\\
&\qquad (acquire_{pc}, \nu).(think, 8/0.01).Client_7\\
Client_8 &\overset{def}{=} (request_{think,save}, \nu).(reply_{think,save}, \nu).\\
&\qquad (acquire_{pc}, \nu).(think, 8/0.01).Client_9\\
Client_9 &\overset{def}{=} (request_{think,read}, \nu).(reply_{think,read}, \nu).\\
&\qquad (acquire_{pc}, \nu).(think, 8/0.01).Client_1
\end{aligned}$$

Server

$$\begin{aligned}
\mathit{Server} &\stackrel{def}{=} (\mathit{request}_{think,visit}, \nu).\mathit{Cache}_1\\
&\quad + (\mathit{request}_{think,buy}, \nu).\mathit{Prepare}_1\\
&\quad + (\mathit{request}_{think,notify}, \nu).\mathit{Notify}_1\\
&\quad + (\mathit{request}_{think,save}, \nu).\mathit{Save}_1\\
\mathit{Cache}_1 &\stackrel{def}{=} (\mathit{acquire}_{ps}, \nu).\\
&\quad \Big[(\mathit{cache}, 0.95 \times 1/0.001).\mathit{Internal}_1\\
&\quad + (\mathit{cache}, 0.05 \times 1/0.001).\mathit{External}_1\Big]\\
\mathit{Internal}_1 &\stackrel{def}{=} (\mathit{acquire}_{ps}, \nu).(\mathit{internal}, 1/0.001).\mathit{Internal}_2\\
\mathit{Internal}_2 &\stackrel{def}{=} (\mathit{reply}_{think,visit}, \nu).\mathit{EndInternal}\\
\mathit{EndInternal} &\stackrel{def}{=} \mathit{Server}\\
\mathit{External}_1 &\stackrel{def}{=} (\mathit{acquire}_{ps}, \nu).(\mathit{external}, 2/0.001).\mathit{External}_2\\
\mathit{External}_2 &\stackrel{def}{=} (\mathit{request}_{external,read}, \nu).(\mathit{reply}_{external,read}, \nu).\\
&\quad (\mathit{acquire}_{ps}, \nu).(\mathit{external}, 2/0.001).\mathit{External}_3\\
\mathit{External}_3 &\stackrel{def}{=} (\mathit{reply}_{think,visit}, \nu).\mathit{EndExternal}\\
\mathit{EndExternal} &\stackrel{def}{=} \mathit{Server}\\
\mathit{Prepare}_1 &\stackrel{def}{=} (\mathit{acquire}_{ps}, \nu).(\mathit{prepare}, 1/0.01).\mathit{ForkPrepare}\\
\mathit{ForkPrepare} &\stackrel{def}{=} (\mathit{fork}_1, \nu).\mathit{EndPrepare}\\
\mathit{EndPrepare} &\stackrel{def}{=} (\mathit{join}_1, \nu).\mathit{Display}_1\\
\mathit{Display}_1 &\stackrel{def}{=} (\mathit{acquire}_{ps}, \nu).(\mathit{display}, 1/0.001).\mathit{Display}_2\\
\mathit{Display}_2 &\stackrel{def}{=} (\mathit{reply}_{think,buy}, \nu).\mathit{EndDisplay}\\
\mathit{EndDisplay} &\stackrel{def}{=} \mathit{Server}\\
\mathit{Notify}_1 &\stackrel{def}{=} (\mathit{acquire}_{ps}, \nu).(\mathit{notify}, 1/0.08).\mathit{EndNotify}\\
\mathit{EndNotify} &\stackrel{def}{=} \mathit{Server}\\
\mathit{Save}_1 &\stackrel{def}{=} (\mathit{acquire}_{ps}, \nu).(\mathit{save}, 2/0.02).\mathit{Save}_2\\
\mathit{Save}_2 &\stackrel{def}{=} (\mathit{request}_{save,write}, \nu).(\mathit{reply}_{save,write}, \nu).\\
&\quad (\mathit{acquire}_{ps}, \nu).(\mathit{save}, 2/0.02).\mathit{Save}_3\\
\mathit{Save}_3 &\stackrel{def}{=} (\mathit{reply}_{think,save}, \nu).\mathit{EndSave}\\
\mathit{EndSave} &\stackrel{def}{=} \mathit{Server}
\end{aligned}$$

Server's Secondary Flows

$$\begin{aligned}
\mathit{Pack}_1 &\stackrel{def}{=} (\mathit{fork}_1, \nu).(\mathit{acquire}_{ps}, \nu).(\mathit{pack}, 1/0.03).\mathit{EndPack}\\
\mathit{EndPack} &\stackrel{def}{=} (\mathit{join}_1, \nu).\mathit{Pack}_1\\
\mathit{Ship}_1 &\stackrel{def}{=} (\mathit{fork}_1, \nu).(\mathit{acquire}_{ps}, \nu).(\mathit{ship}, 1/0.01).\mathit{EndShip}\\
\mathit{EndShip} &\stackrel{def}{=} (\mathit{join}_1, \nu).\mathit{Ship}_1
\end{aligned}$$

FileServer

$$\begin{aligned}
\mathit{FileServer} &\stackrel{def}{=} (\mathit{request}_{think,read}, \nu).\mathit{Read}_1\\
&\quad + (\mathit{request}_{external,read}, \nu).\mathit{Read}_1\\
&\quad + (\mathit{request}_{save,write_1}, \nu).\mathit{Write}'_1\\
\mathit{Read}_1 &\stackrel{def}{=} (\mathit{acquire}_{pfs}, \nu).(\mathit{read}, 1/0.01).\mathit{Read}_2\\
\mathit{Read}_2 &\stackrel{def}{=} (\mathit{reply}_{think,read}, \nu).\mathit{EndRead}\\
&\quad + (\mathit{reply}_{external,read}, \nu).\mathit{EndRead}\\
\mathit{EndRead} &\stackrel{def}{=} \mathit{FileServer}\\
\mathit{Write}'_1 &\stackrel{def}{=} (\mathit{acquire}_{pfs}, \nu).(\mathit{write}_1, 1/0.001).\mathit{Write}'_2\\
\mathit{Write}'_2 &\stackrel{def}{=} (\mathit{reply}_{save,write_1}, \nu).\mathit{EndWrite}'\\
\mathit{EndWrite}' &\stackrel{def}{=} \mathit{Write}''_1\\
\mathit{Write}''_1 &\stackrel{def}{=} (\mathit{acquire}_{pfs}, \nu).(\mathit{write}_2, 3/0.04).\mathit{Write}''_2\\
\mathit{Write}''_2 &\stackrel{def}{=} (\mathit{request}_{write_2,get}, \nu).(\mathit{reply}_{write_2,get}, \nu).\\
&\quad (\mathit{acquire}_{pfs}, \nu).(\mathit{write}_2, 3/0.04).\mathit{Write}''_3\\
\mathit{Write}''_3 &\stackrel{def}{=} (\mathit{request}_{write_2,update}, \nu).(\mathit{reply}_{write_2,update}, \nu).\\
&\quad (\mathit{acquire}_{pfs}, \nu).(\mathit{write}_2, 3/0.04).\mathit{EndWrite}''\\
\mathit{EndWrite}'' &\stackrel{def}{=} \mathit{FileServer}
\end{aligned}$$

Backup

$$\begin{aligned}
\mathit{Backup} &\stackrel{def}{=} (\mathit{request}_{write_2,get}, \nu).\mathit{Get}_1\\
&\quad + (\mathit{request}_{write_2,update}, \nu).\mathit{Update}_1\\
\mathit{Get}_1 &\stackrel{def}{=} (\mathit{acquire}_{pfs}, \nu).(\mathit{get}, 1/0.01).\mathit{Get}_2\\
\mathit{Get}_2 &\stackrel{def}{=} (\mathit{reply}_{write_2,get}, \nu).\mathit{EndGet}\\
\mathit{EndGet} &\stackrel{def}{=} \mathit{Backup}\\
\mathit{Update}_1 &\stackrel{def}{=} (\mathit{acquire}_{pfs}, \nu).(\mathit{update}, 1/0.01).\mathit{Update}_2\\
\mathit{Update}_2 &\stackrel{def}{=} (\mathit{reply}_{write_2,update}, \nu).\mathit{EndUpdate}\\
\mathit{EndUpdate} &\stackrel{def}{=} \mathit{Backup}
\end{aligned}$$

PClient

$$\begin{aligned}
\mathit{PClient}' &\stackrel{def}{=} (\mathit{acquire}_{pc}, \nu).\mathit{PClient}''\\
\mathit{PClient}'' &\stackrel{def}{=} (\mathit{think}, 8/0.01).\mathit{PClient}'
\end{aligned}$$

PServer

$$\begin{aligned}
\mathit{PServer}' &\stackrel{def}{=} (\mathit{acquire}_{ps}, \nu).\mathit{PServer}''\\
\mathit{PServer}'' &\stackrel{def}{=} (\mathit{cache}, 1/0.001).\mathit{PServer}'\\
&\quad + (\mathit{internal}, 1/0.001).\mathit{PServer}'\\
&\quad + (\mathit{external}, 2/0.001).\mathit{PServer}'\\
&\quad + (\mathit{prepare}, 1/0.01).\mathit{PServer}'\\
&\quad + (\mathit{pack}, 1/0.03).\mathit{PServer}' + (\mathit{ship}, 1/0.01).\mathit{PServer}'\\
&\quad + (\mathit{display}, 1/0.001).\mathit{PServer}'
\end{aligned}$$

PFileServer: (cfr. Figure 3)

Complete Layered Queueing Network

$$\begin{aligned}
&\Big(\mathit{Client}[2] \underset{M_1}{\bowtie} (\mathit{Server}[2] \underset{L_1}{\bowtie} \mathit{Pack}_1[2] \underset{L_2}{\bowtie} \mathit{Ship}_1[2])\\
&\quad \underset{M_2}{\bowtie} \mathit{FileServer}[1] \underset{M_3}{\bowtie} \mathit{Backup}[1]\Big)\\
&\quad \underset{M_4}{\bowtie} \Big(\mathit{PClient}[2] \underset{\emptyset}{\bowtie} \mathit{PServer}[2] \underset{\emptyset}{\bowtie} \mathit{PFileServer}[2]\Big),\\
M_1 &= \big\{\mathit{request}_{think,visit}, \mathit{reply}_{think,visit}\, \mathit{request}_{think,buy},\\
&\quad \mathit{reply}_{think,buy}, \mathit{request}_{think,notify}, \mathit{request}_{think,save},\\
&\quad \mathit{reply}_{think,save}\big\}\\
L_1 &= L_2 = \{\mathit{fork}_1, \mathit{join}_1\}\\
M_2 &= \big\{\mathit{request}_{think,read}, \mathit{reply}_{think,read}, \mathit{request}_{external,read}\big\}\\
&\quad \mathit{reply}_{external,read}, \mathit{request}_{save,write_1}, \mathit{reply}_{save,write_1}\big\}\\
M_3 &= \big\{\mathit{request}_{write_2,get}, \mathit{reply}_{write_2,get}, \mathit{request}_{write_2,update},\\
&\quad \mathit{reply}_{write_2,update}\big\}\\
M_4 &= \big\{\mathit{acquire}_{pc}, \mathit{think}, \mathit{acquire}_{ps}, \mathit{cache}, \mathit{internal}, \mathit{external},\\
&\quad \mathit{prepare}, \mathit{pack}, \mathit{ship}, \mathit{display}, \mathit{notify}, \mathit{display}, \mathit{acquire}_{pfs},\\
&\quad \mathit{read}, \mathit{write}, \mathit{get}, \mathit{update}\big\}
\end{aligned}$$