Location-Aware Quality of Service Measurements for Service-Level Agreements

Ashok Argent-Katwala[1], Jeremy Bradley[1], Allan Clark[2], and Stephen Gilmore[2]

[1] Department of Computing, Imperial College, London
[2] LFCS, School of Informatics, University of Edinburgh

Abstract. We add specifications of location-aware measurements to performance models in a compositional fashion, promoting precision in performance measurement design. Using immediate actions to send control signals between measurement components we are able to obtain more accurate measurements from our stochastic models without disturbing their structure. A software tool processes both the model and the measurement specifications to give response time distributions and quantiles, an essential calculation in determining satisfaction of service-level agreements (SLAs).

1 Introduction

Accurate performance analysis is essential to the system design process. A system which does not meet its performance and dependability requirements – crucial parts of its trustworthiness or *performability* [1] – is, in practical terms, as unacceptable as a system which does not meet its correctness requirements. Modern engineered systems are vast and complex and so high-level modelling of these systems is a vital step in determining that they satisfy necessary service-level agreements (SLAs). Our attention here is on the quantitative core of such an SLA, which will typically claim that some percentage of incoming requests will receive a response from the system within a specified time bound.

Computing performance results is a subtle matter. The location of performance measurements in a model can have a dramatic effect on the resulting performance measurement. In this paper, we show how performance measurements, known as *stochastic probes*, can be installed in performance models with increased precision. We show how both the positioning of these probes in the performance model, and the translation of these probes using immediate transitions, improves the reliability of the measurement which results.

Good practice in performance modelling suggests the use of a *compositional approach* [2]. Models are structured by building up co-operations between model components, defining complex models as the composition of smaller sub-models. The leading exemplars of languages supporting compositional performance modelling are *stochastic process algebras* (such as PEPA [2], EMPA [3], the Stochastic π-calculus [4] and SPADES [5]). In these languages *model components* are separate units of functionality which perform stochastically timed activities and can be composed. One way to compose model components P and Q is to require

G. Barthe and C. Fournet (Eds.): TGC 2007, LNCS 4912, pp. 222–239, 2008.

them to co-operate on the activities in the set $\mathcal{K}$, allowing them to proceed independently with any activities not listed in $\mathcal{K}$.

$$P \underset{\mathcal{K}}{\bowtie} Q$$

Models can be hierarchically structured in this way. Below, we require P and Q to co-operate on $\mathcal{K}$ as before, and we also require R and S to co-operate on $\mathcal{M}$. Further, we require the composition of P and Q to co-operate with the composition of R and S on any activities in the set $\mathcal{L}$.

$$(P \underset{\mathcal{K}}{\bowtie} Q) \underset{\mathcal{L}}{\bowtie} (R \underset{\mathcal{M}}{\bowtie} S)$$

In process algebras with multi-way synchronisation this hierarchical co-operation over $\mathcal{L}$ can express co-operation between one of P and Q and one of R and S; or three of these components; or even all four (for activities in $\mathcal{K} \cap \mathcal{L} \cap \mathcal{M}$).

Here model components P, Q, R and S represent parts of the system being modelled and activity sets $\mathcal{K}$, $\mathcal{L}$, and $\mathcal{M}$ list the activities performed by these components in co-operation with others. Compositionality facilitates re-use. In our schematic example above P and S might even be instances of the same class of model component (although configured differently by having different partners to co-operate with, and different co-operation sets to operate under).

Given a hierarchically structured model such as this we can define performance measures of interest by adding *measurement components* which seek to expose important activity sequences so that they may be conveniently measured. One use of these would be to compute response time quantiles used in service-level agreements of the form "97.5% of message sends see an acknowledgement within 600 milliseconds."

Such a measurement component is a *stochastic probe* [6] which can be described directly, as model components are, or more conveniently can be generated from a higher-level description language.

$$\left((P \underset{\mathcal{K}}{\bowtie} Q) \underset{\mathcal{L}}{\bowtie} (R \underset{\mathcal{M}}{\bowtie} S)\right) \underset{\mathcal{N}}{\bowtie} \mathit{Probe}$$

The intention is that a model is not disturbed by the addition of a probe in the sense that all of the activities which could happen previously can still happen, and at the same rate as before. Thus if archiving models and results in an organised store for sharing and re-use [7], models can be stored in a canonical form and measurement components and their associated results can be stored separately from these. The relationship between the model and the probe can also be formally recorded, and made available for later inspection and review.

It is intended that several different probes can be applied to a single model without needing to alter the model and it is even possible that probes are re-used, where a single probe is applied to several different models.

In a modelling language which supports multi-way synchronisation (such as PEPA [2]) probes may observe activities even if those activities are performed by model components in co-operation (for example, an activity from the set $\mathcal{K}$ performed by both P and Q).

As introduced in [6], probes are stateful components which can observe activities, can count, and can change state to remember that an activity has been performed. Using these a modeller can check complex service level agreements such as "97.5% of message sends need two retransmissions or fewer to see an acknowledgement within 600 milliseconds."

However, the position of a probe is that of an external observer. The external observer has a location-ignorant viewpoint. He is unable to distinguish an activity α emanating from P's location from an activity α emanating from S's location. This impedes the expression of many service level agreements which arise naturally. For example, "97.5% of sensor message sends need two retransmissions or fewer to see an acknowledgement from the relay within 600 milliseconds."

In the case where we are interested in the activities of P and not those of S one solution could be to move the probe inside the model so that we can focus on P.

$$\left(\left(\left(P \underset{\mathcal{N}}{\bowtie} Probe^{P}\right) \underset{\mathcal{K}}{\bowtie} Q\right) \underset{\mathcal{L}}{\bowtie} \left(R \underset{\mathcal{M}}{\bowtie} S\right)\right)$$

This would be effective in this case but if instead any of the activities performed by other components (say, S) influence the state of the probe then the probe is at the wrong place in the composed model to observe them. To remedy this we could add another probe to S and have both of these slave probes report to a master which combines their reports appropriately.

$$\left(\left(\left(P \underset{\mathcal{N}}{\bowtie} Probe^{P}\right) \underset{\mathcal{K}}{\bowtie} Q\right) \underset{\mathcal{L}}{\bowtie} \left(R \underset{\mathcal{M}}{\bowtie} \left(S \underset{\mathcal{O}}{\bowtie} Probe^{S}\right)\right)\right) \underset{\mathcal{T}}{\bowtie} Probe^{Master}$$

The addition of these probes is an automated procedure performed on an input model without probes. The modeller need not see the version of the model expanded by the addition of the measurement components and can consider this just to be an intermediate form produced before state-space derivation (in a manner similar to unfolding a coloured Petri net).

The position of $Probe^{P}$ allows it to send to the $Probe^{Master}$ the control message "P performed α" on seeing an activity α performed by P. Similarly the position of $Probe^{S}$ allows it to send to the $Probe^{Master}$ the control message "S performed α" on seeing an activity α performed by S. Model components Q and R could be probed in exactly the same way.

In a purely Markovian process algebra such as PEPA there is a fundamental difficulty with the above design; all activities are timed, and so a rate must be associated with the control messages. The duration of these control messages would then be added to the duration of the model activities occurring in the passage from the start state to the final state. This would interfere with the passage time calculation being made and lead to inaccurate numerical results being produced. We could try to repair this by assigning control messages a rate several orders of magnitude higher than any already in the model but this would not entirely solve the problem because the infinite support of the exponential distribution means that there is a possibility that "fast" control messages are occasionally beaten by "slow" model activities, leading to the master probe being out-of-step with the model description. Even if this problem does not arise the

widely-separated values for the rate constants would very likely lead to stiffness problems in the numerical solution of the underlying Markov chain.

We address this problem by using high-priority immediate actions for the control messages (whereas the process algebra model being probed contains only low-priority exponentially timed activities). Instantaneous control messages flow from the slave probes to the master probe, sending the control signal needed without perturbing the passage-time measurement taking place.

The idea of extending high-level Markovian modelling languages with immediate actions is not new. Stochastic Petri nets were extended to Generalised Stochastic Petri nets in [8] by incorporating immediate transitions and distinguishing between tangible and vanishing states. Neither is the use of immediate actions with stochastic process algebras new. The languages EMPA [3], MoDeST [9], SM-PEPA [10] and SPADES [5] all support immediate actions.

The novelty in the present paper is the introduction of immediate actions in a structured way which facilitates the development of a powerful query language for Markovian models which is an extension of the language proposed in [11]. We first present the ideas from the existing query language then show the location-aware extension together with an example. We have implemented the query language in a new software tool.

The query language which we propose for Markovian models can be used as an alternative to logics such as CSL used in the stochastic model-checking approach [12]. One feature which may be of benefit to users is that our query language offers features such as activity counting and location-identification which cannot be expressed directly in a CSL formula. The technology which underpins both styles is the same: transient analysis of a continuous-time Markov chain.

2 Stochastic Probes

In assessing service level agreements it is often convenient to measure from the observation of one of a set of "start" activities to an occurrence of one of a further set of "stop" activities. For example, $(a{:}\mathsf{start} \mid b{:}\mathsf{start}), c+, (x{:}\mathsf{stop} \mid y{:}\mathsf{stop})$. From this a master probe is generated with two distinct states for *running* and for *stopped* as described in [6]. The probe begins *stopped* and moves to *running* if it observes any of the start activities. Since the master probe must cooperate with the model over the start and stop activities it must be capable of performing these in both states. ("$(a, \top)$" passively observes the timed activity a.)

$$\begin{aligned} Probe^{Master}_{stopped} &\stackrel{\text{def}}{=} (a,\top).Probe^{Master}_{running} + (b,\top).Probe^{Master}_{running} \\ &\quad + (x,\top).Probe^{Master}_{stopped} + (y,\top).Probe^{Master}_{stopped} \\ Probe^{Master}_{running} &\stackrel{\text{def}}{=} (x,\top).Probe^{Master}_{stopped} + (y,\top).Probe^{Master}_{stopped} \\ &\quad + (a,\top).Probe^{Master}_{running} + (b,\top).Probe^{Master}_{running} \end{aligned}$$

The master probe synchronises with the whole model (including the observation probe) over the start and stop activities but not any other activities which the probe may perform, in our case c.

$$(Model \underset{\{a,b,c,x,y\}}{\bowtie} Probe_1^{Obs}) \underset{\{a,b,x,y\}}{\bowtie} Probe_{stopped}^{Master}$$

The start and stop activities are used as communications from the observation probe to the master probe. Whenever an a or b activity is performed the observation probe signals to the master probe to begin measurement and conversely for stop activities.

This will not work for a location-aware probe. The purpose of applying the probe to only a part of the larger model was that the probe could then ignore any of the "start" or "stop" activities performed by other parts of the model with which the current measurement is unconcerned. Instead of cooperating with the master probes over the "start" activities (a and b) and the "stop" activities (x and y), the probe can instead send immediate control messages (*start* and *stop*) to the master probe to say that the activities of interest have been observed. By using immediate actions as the control messages the observation probe may communicate with the master probe in a private manner which also does not affect the model being observed.

$$\begin{aligned}
Probe_1^{Obs} &\stackrel{\text{def}}{=} (a,\top).start.Probe_2^{Obs} + (b,\top).start.Probe_2^{Obs} \\
&\quad + (c,\top).Probe_1^{Obs} \\
&\quad + (x,\top).Probe_1^{Obs} + (y,\top).Probe_1^{Obs} \\
Probe_2^{Obs} &\stackrel{\text{def}}{=} (a,\top).Probe_2^{Obs} + (b,\top).Probe_2^{Obs} \\
&\quad + (c,\top).Probe_3^{Obs} \\
&\quad + (x,\top).Probe_2^{Obs} + (y,\top).Probe_2^{Obs} \\
Probe_3^{Obs} &\stackrel{\text{def}}{=} (a,\top).Probe_3^{Obs} + (b,\top).Probe_3^{Obs} \\
&\quad + (c,\top).Probe_3^{Obs} \\
&\quad + (x,\top).stop.Probe_1^{Obs} + (y,\top).stop.Probe_1^{Obs}
\end{aligned}$$

The master probe is altered so that instead of observing the model (including the observation probe) performing a, b, x and y actions it observes only start and stop communication events.

$$\begin{aligned}
Probe_{stopped}^{Master} &\stackrel{\text{def}}{=} start.Probe_{running}^{Master} + stop.Probe_{stopped}^{Master} \\
Probe_{running}^{Master} &\stackrel{\text{def}}{=} stop.Probe_{stopped}^{Master} + start.Probe_{running}^{Master}
\end{aligned}$$

The names start and stop are labels in the regular expression syntax of probes. Because these now turn into communication signals, the labels can be generalised to include any names that the user wishes. In this way multiple observation probes may be attached to various portions of the model. Their communication signals are distinct labels so these probes avoid name clashes. Generally a control probe will cooperate over the whole model and interpret all of the communication signals from localised observation probes.

3 Location-Aware Stochastic Probes

This example illustrates the need for location-aware probes. The model is that of a simple client server system. The key point is that there are two indistinguishable servers available to respond to each of the three indistinguishable clients and the problem is correctly matching requests and responses.

$$\begin{aligned} Client_{idle} &\stackrel{\text{def}}{=} (request, \lambda).Client_{waiting} \\ Client_{waiting} &\stackrel{\text{def}}{=} (response, \top).Client_{idle} \\ \\ Server_{idle} &\stackrel{\text{def}}{=} (request, \top).Server_{computing} \\ Server_{computing} &\stackrel{\text{def}}{=} (compute, \pi).Server_{responding} \\ Server_{responding} &\stackrel{\text{def}}{=} (response, \rho).Server_{idle} \\ \\ System &\stackrel{\text{def}}{=} Client_{idle}[3] \underset{\mathcal{L}}{\bowtie} Server_{idle}[2] \\ \text{where } \mathcal{L} &= \{request, response\} \end{aligned}$$

Suppose one wishes to measure the expected response time, that is the time taken from a particular client making a request to that client receiving a response. A probe component is added to the model which passively observes all *request* and *response* activities flipping between running and stopped states appropriately. The desired measurement can then be taken to be the expected time for the probe component to be in the running state. So for our model a first attempt at a measurement of response time may be to add the probe in this fashion:

$$System \stackrel{\text{def}}{=} (Client_{idle}[3] \underset{\mathcal{L}}{\bowtie} Server_{idle}[2]) \underset{\mathcal{L}}{\bowtie} Probe \tag{3.1}$$

This global probe over-estimates the performance of the system because it measures the time from *some* client's request to whenever *either* of the servers responds. In particular it may measure the time between one client's *request* and the *response* which corresponds to an earlier *request* performed by another client.

The reason that the global probe does not work as we would expect it is due to the fact that it cannot distinguish between identical actions performed by separate components. Additionally the probe only observes start actions when it is in the stopped state. For this reason when the model performs more than one start action before a stop action is encountered, the probe will still be running.

Figure 1 depicts the error that the response from *Server*[2] to *Client*[2] is paired with the request from *Client*[3] to *Server*[1]. This measurement error occurs due to the use of a location unaware probe.

To fix this problem, the probe can be location-aware. Instead of cooperating with the entire system, the probe cooperates only with a single *Client* process. Writing ($\|$) to denote cooperation over the empty set the system is:

$$System \stackrel{\text{def}}{=} \left((Client_{idle} \underset{\mathcal{L}}{\bowtie} Probe) \parallel Client_{idle}[2]\right) \underset{\mathcal{L}}{\bowtie} Server_{idle}[2] \tag{3.2}$$

The graph in Figure 2 shows the difference in measurement between the local probe from (3.2) and the global probe from (3.1). The graph plots the measured

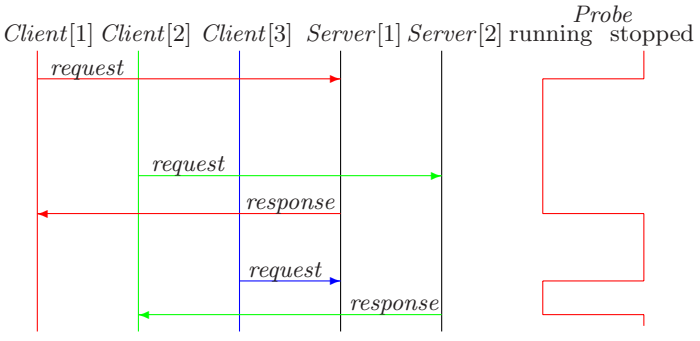

Fig. 1. Diagram showing the trace of a run with a faulty global probe

time since a *request* action against the probability that the probe has cooperated over a *response* action. From this graph the error of the global probe is apparent. The line plotted for the probe is above that of the local probe indicating that the probability of observing a *response* activity is higher.

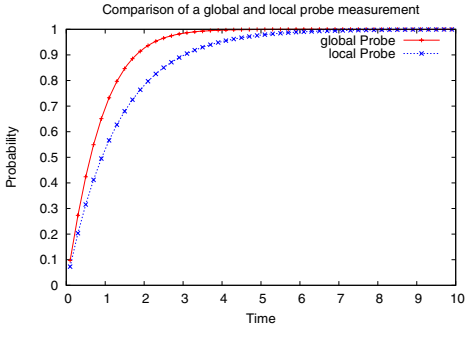

Fig. 2. Graph showing the flawed measurement taken from a global probe versus the true measurement obtained from a local probe

4 Impact on Aggregation

Modelling formalisms founded on Continuous-Time Markov Chains (CTMCs) suffer from the well-known problem of state-space explosion whereby the number of states of the model as a whole may be as large as the product of the number of states of each of the model components. Model *aggregation* [13] battles this state-space growth by exploiting symmetries in the model to reduce the number of states in the state-space. This is done by replacing several *strongly equivalent* states with a canonical representative of them, and adjusting the outgoing rates accordingly. Aggregation based on strong equivalence induces a *lumpable* [14]

partition of the state-space which preserves performance measures. The proof of this result appears in [2]. The definition of the strong equivalence relation is also found there.

Aggregation depends on replication of components in that each component C in an array of N copies of C, $C[N]$, is considered to be interchangeable. With a location-aware measurement component we are able to isolate one of these copies and make it no longer interchangeable with the others. An inevitable consequence of this is that aggregation will now be less productive (because there are now effectively only $N-1$ copies of the component, and so symmetries which existed before have now been broken).

In the worst case, isolating a model component in this way may decrease the profit from aggregation to the point where the model is no longer solvable because its memory requirements exceed those of the machine on which the analysis is taking place. We view this as an inescapable cost of the more accurate identification of model components afforded by location-aware measurement components.

5 Communicating Stochastic Probes

This example expands upon the first to show the need for immediate communication between location-aware probes. We wish to analyse the impact of the breakdown of a server on the response time. In order to measure this our model from before is enhanced with the possibility for servers to break down. Once a server has broken down it must be repaired before it can continue to service client requests.

$$\begin{aligned} Server_{idle} &\stackrel{\text{def}}{=} (request, \top).Server_{responding} \\ &\quad + (break, \kappa).Server_{broken} \\ Server_{responding} &\stackrel{\text{def}}{=} (response, \rho).Server_{idle} \\ Server_{broken} &\stackrel{\text{def}}{=} (repair, \nu).Server_{idle} \\ \\ System &\stackrel{\text{def}}{=} Client_{idle}[3] \underset{\mathcal{L}}{\bowtie} Server_{idle}[2] \\ \text{where } \mathcal{L} &= \{request, response\} \end{aligned}$$

In this example the *Client* processes include a local working activity.

$$\begin{aligned} Client_{idle} &\stackrel{\text{def}}{=} (work, \mu).Client_{requesting} \\ Client_{requesting} &\stackrel{\text{def}}{=} (request, \lambda).Client_{waiting} \\ Client_{waiting} &\stackrel{\text{def}}{=} (response, \top).Client_{idle} \end{aligned}$$

Suppose we wish to determine the response time if the client is ready to make the request when at least one of the servers is currently broken. One way to do this is to insist that the probe observes the *work* activity from the probed client after observing a *break* activity from one of the servers and without observing a *repair* activity. Note that a *repair* activity may take place after the *work* activity has been observed by the probe and the measurement begun.

We require one probe which is attached to a single client, a further probe which is attached to one of the servers, and a master probe which combines the communication messages from the two local probes.

$$\begin{aligned} Clients &\stackrel{\text{def}}{=} \left((Client_{idle} \underset{\mathcal{L}}{\bowtie} Probe^{Client}_{stopped}) \parallel Client_{idle}[2] \right) \\ Servers &\stackrel{\text{def}}{=} \left((Server_{idle} \underset{\mathcal{M}}{\bowtie} Probe^{Server}_{stopped}) \parallel Server_{idle} \right) \\ System &\stackrel{\text{def}}{=} \left((Clients \underset{\mathcal{L}}{\bowtie} Servers) \underset{\mathcal{N}}{\bowtie} Probe^{Master}_{stopped} \right) \\ \text{where } \mathcal{L} &= \{work, response\} \\ \mathcal{M} &= \{break, repair\} \\ \mathcal{N} &= \{clientWork, clientRes, in, out\} \end{aligned}$$

The local server probe is attached to one of the servers and sends a signal to the master probe whenever the local server breaks down or is repaired.

$$\begin{aligned} Probe^{Server}_{stopped} &\stackrel{\text{def}}{=} (break, \top).in.Probe^{Server}_{broken} \\ &\quad + (repair, \top).out.Probe^{Server}_{stopped} \\ Probe^{Server}_{broken} &\stackrel{\text{def}}{=} (repair, \top).out.Probe^{Server}_{stopped} \\ &\quad + (break, \top).in.Probe^{Server}_{broken} \end{aligned}$$

When the local client probe passively observes a *work* activity in one of the clients it sends a communication message to the master probe. Upon observing a *response* activity it sends a message again.

$$\begin{aligned} Probe^{Client}_{stopped} &\stackrel{\text{def}}{=} (work, \top).clientWork.Probe^{Client}_{run} \\ Probe^{Client}_{run} &\stackrel{\text{def}}{=} (response, \top).clientRes.Probe^{Client}_{stopped} \end{aligned}$$

The master probe then receives the communication from the two local probes and connects together the logic to determine whether or not the measurement should begin. It has three states. In the first state, $Probe^{Master}_{stopped}$, it waits for a communication message indicating that one of the servers is broken. When this occurs it moves on to the second state. In the second state, $Probe^{Master}_{waiting}$, there is at least one server broken hence should the probe local to the client send a message indicating that a measurement may begin (that is, the client has performed a *work* action) then the master probe will indeed begin measurement by entering the third state $Probe^{Master}_{running}$. In this state the only message of interest is one from the client probe to indicate that it has observed a *response* activity which causes the measurement to terminate. Note that it is not the case that every *clientWork* activity will cause measurement to start and neither is it the case that every *clientRes* activity will cause measurement to stop.

$$\begin{aligned} Probe^{Master}_{stopped} &\stackrel{\text{def}}{=} in.Probe^{Master}_{waiting} \\ &\quad + clientWork.Probe^{Master}_{stopped} \\ &\quad + clientRes.Probe^{Master}_{stopped} \\ Probe^{Master}_{waiting} &\stackrel{\text{def}}{=} clientWork.start.Probe^{Master}_{running} \end{aligned}$$

$$
\begin{aligned}
& + \ out.Probe^{Master}_{stopped} \\
& + \ clientRes.Probe^{Master}_{waiting} \\
Probe^{Master}_{running} \stackrel{\text{def}}{=} \ & clientRes.stop.Probe^{Master}_{stopped} \\
& + \ out.Probe^{Master}_{running}
\end{aligned}
$$

For the purposes of explanation the probes defined here have been given as though directly written by the user. However in general such probes are specified using a regular expression-like syntax. They are then automatically attached to the model at the appropriate place. To reproduce the full model with the probes attached the following three probe specifications would be given:

1. $Client :: (work : clientWork, response : clientReq)$
2. $Server :: (break : in, repair : out)$
3. $(in, clientWork : start)/out, clientRes : stop$

The first two probes specify a location to which the probe should be attached, *Client* and *Server* respectively. The final probe is the master probe and will be attached to the whole model and hence does not require a location. The syntax $/out$ specifies that the whole of the probe to the left must be observed without observing an *out* signal. Should one occur during the sequence then the probe is reset. We analysed this model with the probes given and with two other configurations. The results are shown in the graph in Figure 3. The line labelled "maybe" is the model analysing from a *clientWork* message to a *clientRes* message regardless of the state of the servers at that time.

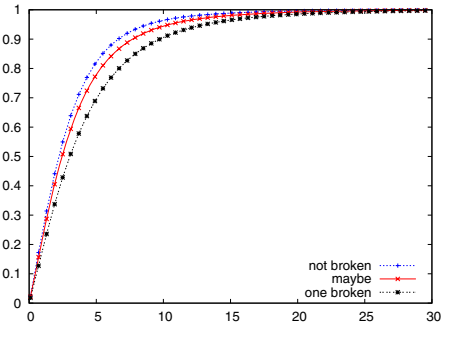

Fig. 3. Graph showing the change of completion of a client's request depending on the state of the servers

6 Worked Example: Wireless Sensor Network

As a worked example, we present a model of a lossy wireless sensor network. The network consists of a set of *SensorBot*s which monitor the environment and

report key events across the network. The bots both take measurements and route traffic from other bots in the network. In routing traffic either from other bots or itself, a bot has a simple send-acknowledge mechanism for sending traffic to a nearest-receiving bot. If an ack is not received, the bot enters a backoff phase before retrying, and repeats this until an ack is received.

$$\begin{aligned}
SensorBot &\stackrel{\text{def}}{=} (monitor, r_{mon}).SensorBot \\
&+ (monitorActive, r_{monA}).SensorBotSend \\
&+ (messageIn, \top).SensorBotRelay \\
&+ (messageIn, \top).SensorBotProcess \\
SensorBotProcess &\stackrel{\text{def}}{=} (ackOut, r_{ack}).(think, r_{think}).SensorBot \\
SensorBotSend &\stackrel{\text{def}}{=} (messageOut, r_{msgOut}).SensorBotWait \\
SensorBotWait &\stackrel{\text{def}}{=} (ackIn, \top).SensorBot \\
&+ (timeout, r_{timeout}).SensorBotRetrySend \\
SensorBotRetrySend &\stackrel{\text{def}}{=} (backoff, r_{backoff}).SensorBotSend \\
&+ (giveup, r_{giveup}).SensorBot \\
SensorBotRelay &\stackrel{\text{def}}{=} (ackOut, r_{ack}).SensorBotSend
\end{aligned}$$

Each *SensorBot* is symmetrically described and is either involved in: monitoring events, *SensorBot*; processing a received event notification from another bot, *SensorBotProcess*; sending a message, *SensorBotSend*; waiting for an ack from another bot that received its message, *SensorBotWait*; resending a message after a backoff period, *SensorBotRetrySend*; or relaying a message across the sensor network, *SensorBotRelay*.

The *SensorBot*s communicate over an unreliable wireless network that comprises a number of channels:

$$\begin{aligned}
UnreliableChannel &\stackrel{\text{def}}{=} (messageOut, \top).UnreliableChannelMsg \\
&+ (ackOut, \top).UnreliableChannelAck \\
UnreliableChannelMsg &\stackrel{\text{def}}{=} (messageIn, r_{netDelay}).UnreliableChannel \\
&+ (messageLose, r_{msgLose}).UnreliableChannel \\
UnreliableChannelAck &\stackrel{\text{def}}{=} (ackIn, r_{netDelay}).UnreliableChannel \\
&+ (messageLose, r_{msgLose}).UnreliableChannel
\end{aligned}$$

A channel can relay a message from one bot to another bot, by picking up a *messageOut* action and transmitting a *messageIn* action to a receiver bot. A similar process transmits acknowledgement messages. What makes this network unreliable is that there is a probability that any given message may be lost, where the probability of loss is determined by:

$$\frac{r_{msgLose}}{r_{msgLose} + r_{netDelay}}$$

Finally, the whole sensor network comprises B bots connected by the unreliable wireless network of C channels, as described by;

$$SensorNet \stackrel{\text{def}}{=} SensorBot[B] \underset{L}{\bowtie} UnreliableChannel[C]$$

where $L = \{ackIn, ackOut, messageIn, messageOut\}$.

This is a simplistic protocol, where it is for instance possible for one bot to acknowledge the message that another bot received. The system probabilistically guards against this, by incorporating a quick timeout mechanism. If the sending bot does not hear an acknowledgement within a short window, it backs off and retries later. If it does hear an acknowledgement, it assumes that this was the response for its message. Given the power constraints involved in sensor networks, this type of simplistic mechanism is not an unreasonable way to conserve sensor battery-life. If guaranteed message sending is required, then a more sophisticated protocol could be deployed.

6.1 Location Probe Measurements

In this model the measurement in which we may be interested is the length of time a sensor can expect to wait for an acknowledgement. This model is distinct from the earlier "Client–Server" style of model in that each sensor acts as both a "Client" and a "Server". Since in this case the response is the acknowledgement that the message has been routed onwards and the sender can continue its monitoring operations. In the traditional "Client–Server" style of model it is clear that as we increase the number of "Client" components without increasing the number of "Server" components the response-time for each individual "Client" should worsen. In the distributed setting of the sensor net, because each additional "Client" (or *SensorBot*) also becomes a "Server" it is less clear how the addition of *SensorBot* components will affect the response-time for each individual *SensorBot*. With each additional *SensorBot* there is a further "Server" which may respond to the individual measured *SensorBot*. However in addition there is an additional "Client" component which may compete not only for the "Server" components but also for the resource components modelled here by the unreliable network channels.

To measure the response-time for a single *SensorBot* component we wish to measure between occurrences of the activity *messageOut* – the *SensorBot* has sent a message to be delivered – and the activity *ackIn* – the *SensorBot* has received an acknowledgement that the message has been relayed/accepted. To achieve this we cannot attach a global-probe component to the entire model as this will not distinguish the occurrences of *messageOut* activity and the *ackIn* activity performed by separate *SensorBot* components. We therefore attach a probe to a single *SensorBot* component. The probe itself waits for an occurence of the *messageOut* activity to start the measurement and an occurrence of the *ackIn* activity to end it. This is written down in our probe language as:

$$SensorBot :: (messageOut : start, ackIn : stop)$$

Figure 4 shows the cumulative distribution functions for the model as we vary the number of *SensorBot* and *UnreliableChannel* components. These

results suggest that increasing the number of *SensorBot* components always improves the response-time. The number of channels may act as a bottleneck in the network and hence increasing the number of channels likewise improves the response-time. Therefore in the "Client–Server" style of model increasing the number of "Servers" is the only way to increase the performance of the system. However in the distributed network increasing either the "peers" or the resources (channels) leads to an improvement in the number of messages relayed.

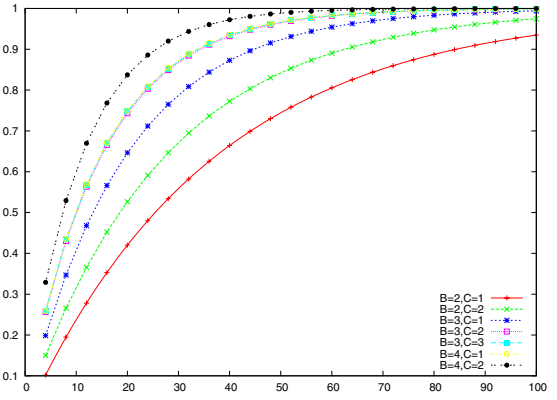

Fig. 4. Graph showing the cdf from 'msgOut' to 'ackIn' for the sensor net model varying the numbers of sensors and channels

7 Design

An extended Markovian process algebra with immediate actions is a step on the way towards the more ambitious goal of an extended process algebra with general distributions. We first explain the relationship between this algebra and immediate actions and then explain the relationship between immediate actions and probes.

7.1 Immediate Actions and SM-PEPA

Semi-Markov PEPA [10] (SM-PEPA), is a version of PEPA that allows general distributions as well as exponential distributions from the standard PEPA model. The syntax for SM-PEPA is given below:

$$P ::= (a^{[n]}, D).P \;\Big|\; P + P \;\Big|\; P \underset{L}{\bowtie} P \;\Big|\; P/L \;\Big|\; A \tag{7.3}$$

where:

$$D ::= \lambda \;\Big|\; \omega : L(s) \tag{7.4}$$

where λ is the standard PEPA exponential rate parameter:

$$\lambda \in \mathbb{R}^+ \cup \{r\top \mid r \in \mathbb{Q}, r > 0\}$$

The action a is annotated with a priority $n \in \mathbb{N}$ (where a larger n indicates a higher priority). SM-PEPA introduces a notion of priority-enabling where an action is priority-enabled only if it is enabled in the normal PEPA sense and there are no higher priority actions that are enabled at the same time. The D variable indicates a duration, either an exponential rate or a weighted general distribution. The general distribution is specified in terms of its Laplace transform for numerical convenience. The weights, ω, are used to select probabilistically between concurrently priority-enabled generally-distributed actions.

The use of priorities in activities (action-duration pairs) is restricted so that within a particular priority level, either Markovian activities are available (containing standard PEPA) or generally-distributed activities are. This prevents the simultaneous racing of exponential and generally-distributed distributions. A detailed semantics for SM-PEPA can be found in [10].

The immediate transition model required for use with stochastic probes can be derived from a subset of SM-PEPA; it uses a similar approach as that used in generalised stochastic Petri nets (GSPNs) [15]. For this purpose only two priority levels are required, level 1 for Markovian activities and level 2 for immediate actions. We use the standard PEPA prefix notation $(a, \lambda).P$ to mean $(a^{[1]}, \lambda).P$ in SM-PEPA and the enhanced immediate prefix notation $(a, \mathit{immediate}).P$ to mean $(a^{[2]}, 1 : 1).P$. This gives each immediate transition equal weight (although we avoid simultaneous enabling of immediate actions in our use of probes here). Where user-defined weighting of immediate transitions is useful, $(a, \omega : \mathit{immediate}).P$ is translated to $(a^{[2]}, \omega : 1).P$. The immediate transition aspect is represented by the Laplace transform, $L(s) = 1$.

7.2 Immediate Actions and Probes

In working with immediate actions together with timed activities we need to clarify how these interact. The first design decision to resolve is with respect to the relative priority of actions and activities.

Priority: Immediate actions have priority over timed activities.

It is necessary to impose this requirement, as in GSPNs, so as to avoid potential problems associated with infinite re-enabling of timed and immediate activities. The priorities are obtained from the mapping to SM-PEPA, described earlier.

The second design decision relates to the names of immediate actions and timed activities.

Separation: Actions and activities have different names.

Concretely, we never have (α, r) and α in the same model. Similarly we never find $(\alpha, \top)$ and α in the same model. Co-operation in PEPA is based on the matching of names and so we have the following consequence from this design decision.

Homogeneity: Actions and activities do not co-operate.

That is, from the semantics of SM-PEPA, we disallow co-operation between immediate actions and timed activities. We use different terms for the two kinds of name-matching, saying that components *co-operate* on timed activities and *synchronise* on immediate actions.

We use immediate actions to report on the occurrence of a timed activity. For this reason timed activities must precede immediate actions.

Pursuit: In each model component every immediate action must be preceded by a timed activity.

We consider Markov models with non-deterministic choice not to be well-specified. This concern has been thoroughly studied previously with generalised stochastic Petri nets and stochastic activity nets [16]. Immediate actions have a default *weight* (of 1) thus $\alpha.P + \beta.Q$ expresses a weighted probabilistic choice between performing action α and continuing as P or performing action β and continuing as Q where each of these outcomes is equally likely. Syntactically $\alpha.P + \beta.Q$ is an abbreviation for $(\alpha, \mathit{immediate}).P + (\beta, \mathit{immediate}).Q$ and a 3:2 weighted choice is written as $(\alpha, 3 : \mathit{immediate}).P + (\beta, 2 : \mathit{immediate}).Q$.

Finally, the purpose of immediate actions in this context of stochastic probes is to send control signals between measurement components in the model. For this reason, we disallow individual occurrences of immediate actions; these must form a synchronisation point between measurement components.

Synchronisation: Each immediate action must be performed as a synchronisation event between two (or more) components.

Immediate actions may not be performed by one component individually. Thus, for example, we will never see $(\tau, \mathit{immediate})$ in a model, because components cannot synchronise on the silent τ action.

It would be possible to avoid the need to use immediate actions, or indeed measurement components entirely, if we altered or rewrote the model to allow a particular passage-time calculation. We are not willing to do this. Customising the model in this way would injure its potential for re-use. Further, making visible at the top level particular start and stop activities at the beginning and end of the passage of interest may require context-sensitive renaming of activities and the introduction of choices between distinguished names, with a corresponding adjustment in the rates at which these activities are performed. Clearly there is great potential for human error here, even assuming that the modeller is willing to customise the model for just the measure of current interest.

Instead of handing the problem of adjusting the model to the modeller, we would rather automate the process to allow instrumentation of the model for location-aware service-level calculations. Introducing immediate actions allows us to do this.

8 Implementation

We have implemented the facility to describe location-aware probes as a companion to the software tool ipc, The Imperial PEPA Compiler [17]. This tool

probe	:=	$location :: R$	A local probe
	|	R	A global probe
location	:=	*processId*	Attach to a single process
	|	$processId[n]$	Attach to an array of processes
	|	*Component*	Detailed component location description
	|	*Cooperation*	Detailed cooperation location description
R	:=	*action*	Observe an action
	|	$R : label$	Send a signal on matching R
	|	R_1, R_2	R_1 followed by R_2
	|	$R_1 \mid R_2$	R_1 or R_2
	|	$R*$	zero or more R
	|	$R+$	one or more R
	|	$R\{n\}$	n R sequences
	|	$R\{m, n\}$	between m and n R sequences
	|	$R?$	one or zero R
	|	R/a	R without observing an a

Fig. 5. The grammar for probe specification in ipc

generates compiled representations of PEPA models in a form suitable for input to the Hydra response-time analyser, the most recent release of the DNAmaca Markov chain analyser [18]. Although we have concentrated here mostly on passage-time computation, ipc also supports the computation of steady-state, transient and counting measures as described in [11].

The new software tool developed for this work is part of the `ipclib` suite, a collection of tools for the specification and evaluation of complex performance measures over Markovian process algebra models. These, and other software tools required, can be downloaded from `http://www.dcs.ed.ac.uk/pepa`.

Probes are defined using a regular-expression-like syntax fully explained in [6]. A probe specification is given by the grammar in Figure 5. The location part specifies where to attach the probe to the model system equation. The *processId* and $processId[n]$ terms specify the location of the probe where that uniquely defines the location, otherwise the *Component* and *Cooperation* syntax are defined in Figure 6.

Where there are a number of choices for a given location, we can pick an individual component or cooperation using the syntax in Figure 6, for instance, by its numeric position. For example, the "third component called P" in the following system is underlined:

$$(P \underset{\mathcal{K}}{\bowtie} P) \underset{\mathcal{L}}{\bowtie} (\underline{P} \underset{\mathcal{M}}{\bowtie} P).$$

The "offering" keyword means that the component, or one of its derivatives, offers the action. We can place a probe at the "component offering *go*, *stop*" to measure the component using some actions the probe expects to see. This lets us use the same measurement description across a range of models.

Component	:=	[*nth*] **component** [*named*] [*offering*] [*coop*]	Choosing a particular component
Cooperation	:=	[*nth*] **cooperation** [*overactions*] [*involving*]	Choosing a particular cooperation
nth	:=	*n***th**	Select a particular numbered match
	|	*n***th to last**	
	|	**last**	
named	:=	**called** *ProcessID*	With a particular name
offering	:=	**offering** [**only**] *Actions*	Performing certain actions
	|	**not offering** *Actions*	
coop	:=	**cooperating***overactions* [**with** *Component*]	In a particular cooperation
overactions	:=	**over** *Actions*	Cooperating over certain actions
involving	:=	**involving** *Component*	Partner component description

Fig. 6. The grammar for probe placement

We can also distinguish between different instances of a component, based on how it cooperates with its neighbours. For example, the "component called P cooperating over b with component called Q" is underlined: $(P \underset{\{a\}}{\bowtie} Q) \underset{L}{\bowtie} (Q \underset{\{b\}}{\bowtie} \underline{P})$.

9 Conclusions

By adding location-awareness to probe specifications, we give the performance modeller the flexibility to identify model components within the model for selective instrumentation. We have shown that this can have a marked effect on the results produced when compared with an approach using only a single external observer, as used in previous work. By enhancing the probe translation to use immediate transitions, we can capture the response time of interest exactly with no introduction of error from the measurement activities of the probe.

In adding these features we have found it necessary to increase the expressiveness of the probe specification language. In doing this we have endeavoured to maintain a simple language syntax. The more straightforward the language which can be used to describe service-level agreements, the lower the barrier to entry to their use, allowing practitioners to access sophisticated performance evaluation technology and apply it at low cost. Our efforts here have been to design a concise, yet clear, mechanism for adding measurement components to model components in a way that improves the precision of the measurement specification and the accuracy of the result.

Acknowledgements. Ashok Argent-Katwala and Jeremy Bradley are supported by PerformDB, under EPSRC grant, EP/D054087/1. Allan Clark and Stephen Gilmore are supported by the EU FET-IST Global Computing 2 project SENSORIA ("Software Engineering for Service-Oriented Overlay Computers" (IST-3-016004-IP-09)). The Hydra response-time analyser was developed by Will Knottenbelt and Nick Dingle of Imperial College, London.

References

1. Meyer, J.F.: On evaluating the performability of degradable computing systems. IEEE Transactions on Computers C-29(8), 720–731 (1980)
2. Hillston, J.: A Compositional Approach to Performance Modelling. Cambridge University Press, Cambridge (1996)
3. Bernardo, M., Gorrieri, R.: A tutorial on EMPA: A theory of concurrent processes with non-determinism, priorities, probabilities and time. Theoretical Computer Science 202, 1–54 (1998)
4. Priami, C.: Stochastic π-Calculus. The Computer Journal 38(6), 578–589 (1995)
5. Strulo, B., Harrison, P.G.: Spades – A process algebra for discrete event simulation. J. Logic Computation 10(1), 3–42 (2000)
6. Argent-Katwala, A., Bradley, J.T., Dingle, N.J.: 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, Redwood Shores, California, USA, pp. 49–58. ACM Press, New York (2004)
7. AESOP performance modelling group: PerformDB performance model database, Imperial College London (2007), `http://performdb.org`
8. Marsan, M.A., Conte, G., Balbo, G.: A class of generalised stochastic Petri nets for the performance evaluation of multiprocessor systems. ACM Transactions on Computer Systems 2(2), 93–122 (1984)
9. D'Argenio, P.R., Hermanns, H., Katoen, J.-P., Klaren, R.: MoDeST – A modelling and description language for stochastic timed systems. In: de Luca, L., Gilmore, S. (eds.) PROBMIV 2001, PAPM-PROBMIV 2001, and PAPM 2001. LNCS, vol. 2165, pp. 87–104. Springer, Heidelberg (2001)
10. Bradley, J.T.: Semi-Markov PEPA: Modelling with generally distributed actions. International Journal of Simulation 6(3-4), 43–51 (2005)
11. Argent-Katwala, A., Bradley, J.T.: Functional performance specification with stochastic probes. In: Horváth, A., Telek, M. (eds.) EPEW 2006. LNCS, vol. 4054, pp. 31–46. Springer, Heidelberg (2006)
12. Kwiatkowska, M., Norman, G., Parker, D.: Stochastic model checking. In: Bernardo, M., Hillston, J. (eds.) SFM 2007. LNCS, vol. 4486, pp. 220–270. Springer, Heidelberg (2007)
13. Gilmore, S., Hillston, J., Ribaudo, M.: An efficient algorithm for aggregating PEPA models. IEEE Transactions on Software Engineering 27(5), 449–464 (2001)
14. Kemeny, J., Snell, J.: Finite Markov Chains. Van Nostrand (1960)
15. Marsan, M.A., Balbo, G., Conte, G., Donatelli, S., Franceschinis, G.: Modelling with Generalized Stochastic Petri Nets. John Wiley, Chichester (1995)
16. Deavours, D.D., Sanders, W.H.: An efficient well-specified check. In: Proceedings of PNPM 1999: the 8th International Workshop on Petri Nets and Performance Models, Zaragoza, Spain, IEEE Computer Society Press, Los Alamitos (1999)
17. Bradley, J., Dingle, N., Gilmore, S., Knottenbelt, W.: Derivation of passage-time densities in PEPA models using ipc: The Imperial PEPA Compiler. In: Kotsis, G. (ed.) Proceedings of the 11th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunications Systems, University of Central Florida, October 2003, pp. 344–351. IEEE Computer Society Press, Los Alamitos (2003)
18. Knottenbelt, W.: Generalised Markovian analysis of timed transition systems. Master's thesis, University of Cape Town (1996)