2012 IEEE 20th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems

Performance Modeling of Design Patterns for Distributed Computation

Ronald Strebelow*, Mirco Tribastone†, Christian Prehofer*
*Fraunhofer Institute for Communication Systems ESK
Munich, Germany
{ronald.strebelow,christian.prehofer}@esk.fraunhofer.de
†Department for Informatics
Ludwig-Maximilians-Universität Munich,Germany
tribastone@pst.ifi.lmu.de

***Abstract*—In software engineering, design patterns are commonly used and represent robust solution templates to frequently occurring problems in software design and implementation. In this paper, we consider performance simulation for two design patterns for processing of parallel messaging. We develop continuous-time Markov chain models of two commonly used design patterns, Half-Sync/Half-Async and Leader/Followers, for their performance evaluation in multi-core machines. We propose a unified modeling approach which contemplates a detailed description of the application-level logic and abstracts away from operating system calls and complex locking and networking application programming interfaces. By means of a validation study against implementations on a 16-core machine, we show that the models accurately predict peak throughputs and variation trends with increasing concurrency levels for a wide range of message processing workloads. We also discuss the limits of our models when memory-level internal contention is not captured.**

***Keywords*-Design patterns; performance models; Half-Sync/ Half-Async; Leader/Followers; multi-core systems.**

I. Introduction

In software engineering, design patterns are commonly used and represent robust solution templates to frequently occurring problems in software design and implementation. Initially proposed for object-oriented systems [1], pattern catalogues exist for specialized contexts such as service-oriented architectures [2] and enterprise applications [3].

Motivated by the increasing availability and pervasiveness of multi-core systems, we focus in this paper on design patterns for distributed computation [4]. We consider a typical scenario where a multi-threaded server application is deployed on a multi-core machine. Requests from clients which arrive through network interfaces must be processed by the server; successively, a response message must be prepared and sent to the clients. Design patterns help in the implementation of mechanisms for parallel message processing, providing templates that harness the availability of multiple independent CPUs whilst preserving correctness of access to shared resources.

Even if two distinct implementations are both functionally correct, their performance may be different. There are many factors that may affect this, for instance the concurrency levels deployed (i.e., how many threads are running), the use of a particular version of an operating system, or a specific application programming interface (API).

The goal of this paper is early-stage performance prediction of variants of a software system which considers different design patterns and varying concurrency levels. Our approach is to build models based on continuous-time Markov chains (CTMCs), where the application-level logic is modeled in detail, whereas lower-level artifacts such as hardware specifics and operating-system details are abstracted away. Specifically, these are incorporated into the model by means of service rates associated with operations that involve system calls.

As a case study, we consider two of the most prominent design patterns for distributed computation, namely Half-Sync/Half-Async (hereafter abbreviated as HSHA, see [5]) and Leader/Followers (LF, see [6]). They differ mainly in their distribution of tasks to threads. For instance, LF uses a pool of identical threads which have to synchronize their access to the input streams. In contrast, HSHA has a pipeline-like architecture. It uses a dedicated thread to retrieve incoming messages and forwards them to the remaining threads which process these messages. The performance of both patterns does not solely depend on their architecture, i.e., usage of locks or sharing of data, but also on the availability of efficient APIs for notifications by the operating system about new messages. In this paper, we analyze the well-known `epoll` API (see, e.g., [7]), which is a Linux mechanism to retrieve such notifications from a pool of open network connections through a single system call. With proper modifications, however, other APIs can in principle be modeled such as the POSIX `select` and Windows' `WaitForMultipleObjects`.

We validate the models against measurements taken from a real system running on a machine using up to 16 cores. We carry out sensitivity analysis with respect to varying message processing workloads and increasing concurrency levels. Interestingly, the models can be calibrated using estimates of service rates measured from the systems with the lowest

1526-7539/12 $26.00 © 2012 IEEE
DOI 10.1109/MASCOTS.2012.37

concurrency levels (i.e., with the smallest number of cores used by the application), thus showing a considerable degree of a predictive power. Importantly, this exercise also shows that the level of abstraction adopted is sufficient to keep the model relatively simple, whilst capturing the essential performance characteristics of the system under study. In particular, we are able to predict the performance increase and the expected peak throughputs for both patterns with different loads. We also discuss the limits of our approach: for higher concurrency levels than those yielding peak throughput, the model becomes less accurate because it does not predict performance degradation, yielding a plateau instead. We find that this is due to additional hardware and operating system effects, i.e., contention between threads which are not explicitly captured in the model.

Related work: Performance evaluation of design patterns for distributed systems has received some attention in the past. Recently, a measurement-based study has considered the performance impact of variants of `epoll` on HSHA and LF [8]. Some other empirical investigations consider specific applications such as web servers [9] or CORBA [10], but are not concerned with modeling and prediction. Instead, using a UML description, Gomaa and Menascé have presented queueing models for a comparative assessment of synchronous and asynchronous forms of communication [11]. However, the model results are not compared against real data. Parametrization and validation are also missing in [12] where performance models of design patterns are specified using layered queueing networks [13]. In an analogous fashion, in [14] the authors discuss the performance evaluation of patterns for service-oriented architectures.

Our work is also similar in spirit to [15], which provides a uniform framework based on rewriting logics for the quantitative verification of design patterns for tackling denial-of-service attacks. Here, we also consider models for distinct patterns, given however with the same level of abstraction, but for the purposes of performance prediction. Specifically, the main goal is to provide forms of sensitivity analysis that permit to identify the optimal operating conditions with respect to the concurrency levels devoted to a certain service. This is a universal research goal in multi-core systems, which goes beyond application-level software applications; for instance, predictive models (for energy consumption) have been developed for multi-core graphical processing units [16].

Structure of this paper: The rest of this paper is organized as follows. Section II introduces the `epoll` API, where emphasis is given to the implementation aspects that are most crucial for the development of the performance model. The design patterns HSHA and LF are briefly described in Section III. The models are described in detail in Section IV, whereas Section V presents a validation study. Finally, concluding remarks are given in Section VI.

II. A Case Study of EPOLL

The Linux-specific `epoll` API monitors, in kernel space, a pool of network connections used by a user-level application. A *ready list* is maintained which is populated with entries related to the occurrences of events on any of the monitored connections, such as packet reception or shutdown by the remote peer. Later, the application can pick up all or a subset of these entries, depending on how many entries the application requests.

The API consists of the following system calls. `epoll_create` sets up all kernel-related data structures and returns a handle to those, which is used in subsequent calls. These data structures are not shared among applications but are private. This function will not be modeled subsequently because it is invoked only once in the whole application lifetime. `epoll_ctl` is used to modify the set of monitored network connections. Modifications include adding and removing network connections to/from the pool, and altering the set of events for each registered network connection that the application is interested in. `epoll_wait` copies the ready list to user space, for the application to process connection-related events.

The `epoll` API is often used in conjunction with an option that *deactivates* a network connection if events are reported by `epoll_wait`. This avoids that one event is reported several times, thus avoiding that multiple threads attempt to process the same event. The application will not be able to obtain further events from this network connection before it is *reactivated* explicitly, an operation which is performed by calling `epoll_ctl`. We use `epoll` with the aforementioned deactivation option for our pattern implementations.

To be able to model the interactions between the various threads more precisely, we describe `epoll_ctl` and `epoll_wait` in detail. The pool of managed network connections is maintained as a red-black tree (rb-tree). When `epoll_ctl` is called to reactivate a network connection, an instance-wide mutex (hereafter called *epoll lock*) is used to protect all changes against corruption from concurrent access. First the network connection is searched within the rb-tree. If found, it is reenabled by setting a flag which states that the connection shall be considered for subsequent `epoll_wait` calls. Afterwards, the connection is checked for pending events in which case an entry is added to the ready list. A call to `epoll_wait` first checks if the ready list is non-empty, in which case the aforementioned `epoll` lock is acquired and `epoll_wait` starts parsing the ready list. Each entry in the ready list is copied to user space. Finally, the lock is released. Based on this description, we summarize the main facts which affect the performance of an application that uses `epoll`.

(i) `epoll_wait` and `epoll_ctl` share a lock, therefore queueing effects arise when two distinct threads wish

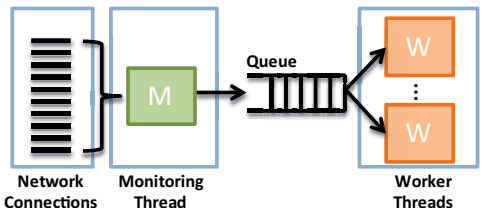


Figure 1: Components of HSHA: The monitoring thread monitors a set of network connections using `epoll`; occurred events are passed to worker threads.

to perform those operations. (ii) The service time of `epoll_wait` depends on the number of network connections with pending events. (iii) The service time of `epoll_ctl` depends logarithmically on the number of network connections the application has currently open, due to the rb-tree. It also depends on whether the connection has pending events. In this paper, we keep the number of open connections constant. Also, since we use a request-response benchmark with at most one request in flight, reactivation will never find pending events. Thus, the average service time of `epoll_ctl` can be assumed to be constant.

III. Software Design Patterns

A. Half-Sync/Half-Async

HSHA [5], depicted in Figure 1, divides the task of event processing into two subtasks: obtaining events from the system and processing them. Both tasks are executed in different threads which communicate via a global queue.

For the first task, one thread, hereafter called the *monitoring thread*, is responsible for handling a number of network connections by means of a suitable API mechanisms, `epoll` in our case. Whenever an event occurs, it is reported to the monitoring thread. The implementation of our mechanism caches as many such events as possible. Both patterns then access these one by one. Only if the cache becomes empty is a new call to the `epoll` API performed. If an event is available, the monitoring thread puts an event notification into the queue. Each notification describes the event and the network connection on which it occurred.

Event processing can be executed by any number of worker threads. Each such thread reads event notifications from the queue, thus if the queue is empty the thread goes into wait. After fetching a notification, the thread starts processing the described event according to the logic of the user-level application. In our case study, processing consists of receiving a request message from the source network connection and reactivating it afterwards (see Section II). Then, the request is processed by busy waiting for a predefined period of time, to model CPU workload, and a response is sent to the client.

Disregarding the network connections, all threads influence each other in two ways: On the one hand, all threads have to access the global queue and therefore have to synchronize. The monitoring thread is not prioritized but has

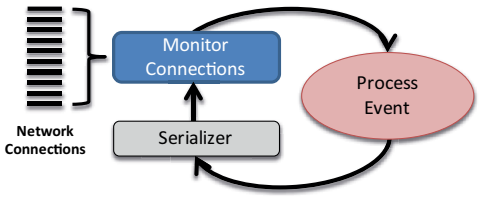


Figure 2: Execution cycle of threads used by LF: Only one thread may monitor the network connection (leading), other threads either process previously obtained events (following) or wait on the serializer to become the leader.

to wait for a chance to put an event notification. On the other hand, threads must synchronize when either reactivating a network connection or obtaining new events, since both are performed using the `epoll` API.

B. Leader/Followers

The LF pattern uses a pool of threads which all execute the same task [6]. This task is shown in Figure 2. As for HSHA, it consists of monitoring network connections and processing occurred events. Due to limitations of the mechanism, only one thread at a time, the *leader thread*, is allowed to monitor the network connections. When events are available in the cache, the leader thread takes one of them and activates another thread which shall resume monitoring. That thread takes one of the remaining events in the cache or, if the cache is empty, restarts monitoring of the connections using `epoll_wait`.

Event processing, on the contrary, is done in parallel. Like HSHA, processing consists of reading the received request, reactivating the source connection, performing some workload, and sending a response. After event processing is finished the thread becomes idle. In this state the thread waits to become the leader and to fetch a new event for processing.

Since each thread executes the same task, all threads also share the same data structures. Therefore the data locality is worse compared to HSHA. On the other hand, since no queue is needed, there is less overhead due to serialization.

IV. Performance Models

A. Mathematical preliminaries

Hereafter, we consider continuous-time Markov chains (CTMCs) with a state descriptor, denoted by x, y, ..., which is a vector of nonnegative integers of length d. Each element, denoted by x_i, with $i \in I$ where I is an index set, is intended to indicate the population of some model component in that state, and d denotes the number of distinct kinds of objects in the model. We use δ_i to indicate a vector of zeros of length d, where the element indexed by $i \in I$ is set to one. For a function $f(x)$ of a state, we also consider the indicator function $H(f(x))$ defined as 1 if $f(x) > 0$ and 0 otherwise.

The infinitesimal generator of the chain is characterized by M *generating functions*, denoted by ϕ_k, $1 \leq k \leq M$, $\phi_k : \mathbb{Z}^d \to \mathbb{R}$, each associated with a *jump vector* $l_k : \mathbb{Z}^d \to \mathbb{Z}^d$. For a state x, $\phi_k(x)$ gives the transition rate due to some action occurring, whereas the jump vector $l_k(x)$ indicates the changes in the population levels in that state for that transition. Let $\hat{x} \in \mathbb{N}_0^d$ be the initial state of the chain.

B. Half-Sync/Half-Async model

The CTMC for the HSHA design pattern has the state descriptor in the form

$$x = (x_c, x_m, x_q, x_t, x_l, x_s, x_b), \tag{1}$$

where

$$\begin{aligned}
x_c &= (x_{c,1}, x_{c,2}, x_{c,3}, x_{c,4}), && \text{[clients]}\\
x_m &= (x_{m,1}, \ldots, x_{m_6}), && \text{[monitor, i.e., I/O thread]}\\
x_q &= (x_{q,1}, x_{q,2}), && \text{[shared queue]}\\
x_t &= (x_{t,1}, \ldots, x_{t,9}), && \text{[worker threads]}\\
x_s &= (x_{s,1}, x_{s,2}), && \text{[epoll lock]}\\
x_b &= (x_{b,1}, x_{b,2}), && \text{[shared queue lock]}
\end{aligned}$$

with the following interpretation:

- $x_{c,1}$: connections ready for `epoll_wait`, i.e., contained in `epoll`'s ready list;
- $x_{c,2}$: cached connections, ready to be handled;
- $x_{c,3}$: connections with pending request which is ready for processing;
- $x_{c,4}$: connections waiting for a response;
- $x_{m,1}$: monitor acquires `epoll` lock;
- $x_{m,2}$: monitor calls `epoll_wait`;
- $x_{m,3}$: monitor releases `epoll` lock;
- $x_{m,4}$: monitor acquires lock to shared queue;
- $x_{m,5}$: monitor puts a socket descriptor into queue;
- $x_{m,6}$: monitor releases queue lock;
- $x_{q,1}$: empty places in the shared queue;
- $x_{q,2}$: busy places in the shared queue;
- $x_{t,1}$: idle worker threads (waiting to obtain a lock to shared queue);
- $x_{t,2}$: worker threads popping a socket descriptor off shared queue;
- $x_{t,3}$: worker threads releasing lock to shared queue;
- $x_{t,4}$: worker threads accepting a request from clients;
- $x_{t,5}$: worker threads obtaining `epoll` lock;
- $x_{t,6}$: worker threads reactivating socket descriptor;
- $x_{t,7}$: worker threads releasing `epoll` lock;
- $x_{t,8}$: worker threads performing computation;
- $x_{t,9}$: worker threads responding to clients;
- $x_{s,1}$: `epoll` lock free, if $x_{s,1} \neq 0$;
- $x_{s,2}$: `epoll` lock busy, if $x_{s,2} \neq 0$;
- $x_{b,1}$: shared queue lock free, if $x_{b,1} \neq 0$;
- $x_{b,2}$: shared queue lock busy, if $x_{b,2} \neq 0$.

The initial state for this model is assumed to be a vector of zeros where

$$\begin{aligned}
\hat{x}_{c,1} &= 512, & \hat{x}_{m,1} &= 1, & \hat{x}_{q,1} &= 512,\\
\hat{x}_{t,1} &= N_x, & \hat{x}_{s_1} &= 1, & \hat{x}_{b_1} &= 1,
\end{aligned}$$

that is, there is a closed workload of 512 connections, a single monitor thread, an initially empty unbounded buffer (here it is set to the maximum number of connections), N_x worker threads, and locks for `epoll` and the shared queue which are initially free.

The generating functions and their corresponding jump vectors are defined thus.

Monitor acquires `epoll` lock:

$$\begin{aligned}
\phi_1(x) &= \frac{1}{x_{m,1} + x_{t,5}} r_l x_{m,1} x_{s,1} H(x_{m,1} + x_{t,5})\\
l_1(x) &= -x_{c,1}\delta_{c,1} + x_{c,2}\delta_{c,2} - \delta_{m,1} + \delta_{m,2} - \delta_{s,1} + \delta_{s,2}.
\end{aligned}$$

The fraction $(x_{m,1}+x_{t,5})^{-1}$ ensures that the rate for locking is fixed to r_l when there are competing threads trying to acquire it (compare to ϕ_{11}). The product $x_{m,1}x_{s,1}$ can be shown to always evaluate to either 0 or 1, thus effectively acting as a boolean guard which does not allow the transition if the monitor is such that $x_{m,1} \neq 1$ and the lock is already taken. This modeling template will be used quite extensively throughout this section. Furthermore notice that, in this model, acquiring the lock involves "freezing" the number of connections which will be handled by the subsequent call to `epoll_wait` (see ϕ_2 below). This is captured by zeroing the population $x_{c,1}$ and adding that to $x_{c,2}$.

Call to `epoll_wait`:

$$\phi_2(x) = (r_e/x_{c,2}) x_{m,2} x_{s,2} H(x_{c,2}),\ l_2(x) = -\delta_{m,2} + \delta_{m,3}.$$

The rate r_e is associated with a single socket descriptor handled during that call; the total rate for `epoll_wait` is assumed to be inversely proportional to the total number of descriptors to be returned.

Monitor releases `epoll` lock:

$$\phi_3(x) = r_l x_{m,3} x_{s,2},\ l_3(x) = -\delta_{m,3} + \delta_{m,4} - \delta_{s,2} + \delta_{s,1}.$$

Monitor acquires lock to shared queue:

$$\begin{aligned}
\phi_4(x) &= \frac{1}{x_{m,4} + x_{t,1}} r_l x_{m,4} x_{b,1} H(x_{m,2})\\
l_4(x) &= -\delta_{m,4} + \delta_{m,5} - \delta_{b,1} + \delta_{b,2}.
\end{aligned}$$

Monitor puts a socket descriptor into queue:

$$\begin{aligned}
\phi_5(x) &= r_p x_{m,5} x_{b,2},\\
l_5(x) &= -\delta_{c,2} + \delta_{c,3} - \delta_{m,5} + \delta_{m,6} - \delta q_1 + \delta_{q,2}.
\end{aligned}$$

This transition has the effect of updating the state of the connection, whose request is now ready to be processed through a worker thread, if any is available; changing the

state of the monitor which may subsequently release the lock to the shared queue (see below); moving a socket handle into the shared queue, with rate r_p, to be fetched by a worker thread.

Monitor releases lock to shared queue:

$$\phi_6(x) = r_l x_{m,6} x_{b,2}$$
$$l_6(x) = -\delta_{m,6} + \delta_{b,1} - \delta_{b,2} + \begin{cases} \delta_{m,1} & \text{if } x_{c,2} = 0, \\ \delta_{m,4} & \text{otherwise.} \end{cases}$$

Upon this transition, the lock is released and the monitor changes state. The case distinction considers the situation whether there are socket descriptors already returned from a previous `epoll_wait` call (and stored in $x_{c,2}$). If that is not the case, the monitor returns to state $x_{m,1}$ where it is willing to do a new `epoll_wait`.

Worker thread acquires lock to shared queue:

$$\phi_7(x) = \frac{1}{x_{m,4} + x_{t,1}} r_l x_{t,1} x_{b,1} H(x_{q,2})$$
$$l_7(x) = -\delta_{t,1} + \delta_{t,2} - \delta_{b,1} + \delta_{b,2}.$$

The case distinction ensures that the lock is not acquired if the shared queue is empty.

Worker thread pops socket descriptor off shared queue:

$$\phi_8(x) = r_g x_{t,2} x_{b,2}, \quad l_8(x) = -\delta_{t,2} + \delta_{t,3} + \delta_{q,1} - \delta_{q,2},$$

where r_g is the rate for an individual removal of an item from the queue.

Worker thread releases lock to shared queue:

$$\phi_9(x) = r_l x_{t,3} x_{b,2}, \quad l_9(x) = -\delta_{t,3} + \delta_{t,4} + \delta_{b,1} - \delta_{b,2}.$$

Worker thread accepting a request from client:

$$\phi_{10}(x) = r_i \min(x_{c,3}, x_{t,4}), l_{10}(x) = -\delta_{c,3} + \delta_{c,4} - \delta_{t,4} + \delta_{t,5},$$

where r_i is the transfer rate of an individual message.

Worker thread acquiring `epoll` lock:

$$\phi_{11}(x) = \frac{1}{x_{m,1} + x_{t,5}} r_l x_{t,5} x_{s,1} H(x_{m,1} + x_{t,5})$$
$$l_{11}(x) = -\delta_{t,5} + \delta_{t,6} - \delta_{s,1} + \delta_{s,2}.$$

Notice the similarity with ϕ_1, which gives the rate for the (competing) monitor thread. If there are $x_{t,5}$ worker threads and one monitor thread willing to acquire the lock, then the probability that a monitor thread acquires the lock is $x_{t,5}/(x_{t,5} + 1)$.

Worker thread reactivating a socket descriptor:

$$\phi_{12}(x) = r_a x_{s,2}, \quad l_{12}(x) = -\delta_{t,6} + \delta_{t,7},$$

where r_a is the reactivation. Its parametrization will be discussed in some detail in Section V-A.

Worker thread releasing `epoll` lock:

$$\phi_{13}(x) = r_l x_{t,7} x_{s,2}, \quad l_{13}(x) = -\delta_{t,7} + \delta_{t,8} + \delta_{s,1} - \delta_{s,2}.$$

Worker thread performing computation:

$$\phi_{14}(x) = r_c x_{t,8}, \quad l_{14}(x) = -\delta_{t,8} + \delta_{t,9},$$

where r_c is an estimate of the computation cost per single request. The validation of this model in Section V-B will consider a range of values for this rate to explore the system behavior across different utilization levels.

Worker thread responding to client:

$$\phi_{15}(x) = r_o \min(x_{c,4}, x_{t,9}), \; l_{15}(x) = \delta_{c,1} - \delta_{c,4} - \delta_{t,1} + \delta_{t,9}.$$

This is analogous to ϕ_{10}, however the rate for the response r_o is in general different from that for the request r_i.

C. Leader/Followers model

In the following, we use y to denote the state descriptor in the LF model, φ its generating functions, and e the corresponding jump vectors. We define y as

$$y = (y_c, y_t, y_s, y_l), \tag{2}$$

where $y_c = (y_{c,1}, \ldots, y_{c,4})$ and $y_s = (y_{s,1}, y_{s,2})$ are defined as x_c and x_s, respectively, and $y_t = (y_{t,1}, \ldots, y_{t,11})$ has the following interpretation:

- $y_{t,1}$: idle threads (waiting to become the leader);
- $y_{t,2}$: leader thread trying to consume a socket descriptor;
- $y_{t,3}$: leader thread calling `epoll_wait`;
- $y_{t,4}$: leader thread releasing lock on `epoll`;
- $y_{t,5}$: thread releasing leader lock;
- $y_{t,6}$: threads accepting requests from clients;
- $y_{t,7}$: treads acquiring lock on `epoll`;
- $y_{t,8}$: thread reactivating socket descriptor;
- $y_{t,9}$: thread releasing lock on `epoll`;
- $y_{t,10}$: threads performing computation;
- $y_{t,11}$: threads responding to clients.

Finally, $y_l = (y_{l,1}, y_{l,2})$ is the state descriptor for the leader lock. The initial state $\hat{y}$ of the LF model is assumed to be a vector of zeros with

$$\hat{y}_{c,1} = 512, \; \hat{y}_{t,1} = N_y, \; \hat{y}_{s,1} = 1, \; \hat{y}_{l,1} = 1.$$

The generating functions are defined thus.

Idle threads acquiring leader lock:

$$\varphi_1(y) = r_l y_{l,1} H(y_{t,1}), \quad e_1(y) = -\delta_{t,1} + \delta_{t,2} - \delta_{l,1} + \delta_{l,2}$$

Leader thread acquiring lock on `epoll`:

$$\varphi_2(y) = \begin{cases} \frac{1}{y_{t,2} + y_{t,7}} r_l y_{t,2} y_{l,1} y_{s,1} & \text{if } x_{c,1} > 0, x_{c,2} = 0, \\ & \quad y_{t,2} + y_{t,7} > 0, \\ 0 & \text{otherwise,} \end{cases}$$
$$e_2(y) = -\delta_{t,2} + \delta_{t,3} - \delta_{s,1} + \delta_{s,2}.$$

As with the HSHA model, the denominator $y_{t,2} + y_{t,7}$ represents the total population of threads who is competing for acquiring the lock on `epoll`. This transition is enabled if no socket descriptor is still cached and `epoll`'s ready list (see Section II) is non-empty.

Leader thread consuming a socket descriptor from cache, i.e., previously returned by `epoll_wait`*:*

$$\varphi_3(y) = r_t y_{t,2} y_{l,1} H(x_{c,2}),\ e_3(y) = -\delta_{c,2}+\delta_{c,3}-\delta_{t,2}+\delta_{t,5}.$$

Notice that φ_2 and φ_3 affect $y_{t,2}$, however in no state are both transitions simultaneously enabled. The model is such that a call to `epoll_wait` will be made only if there are no pending connections, $x_{c,2} = 0$; otherwise one of them will be preferably consumed, with rate r_t.

Leader thread calling `epoll_wait`*:*

$$\begin{aligned}\varphi_4(y) &= (r_e/x_{c,1}) y_{t,3} y_{s,1} y_{l,1} H(x_{c,1})\\ e_4(y) &= -y_{c,1}\delta_{c,1} + (y_{c,1}-1)\delta_{c,2} + \delta_{c,3} - \delta_{t,3} + \delta_{t,4}.\end{aligned}$$

This transition will affect the state of the leader thread which will then release the `epoll` lock. Furthermore, all connections but one will be cached, and one will be immediately handled by the leader when it has released all locks.

Thread releasing `epoll` *lock:*

$$\varphi_5(y) = r_l y_{t,4} x_{s,2} x_{l,2},\quad e_5(y) = -\delta_{t,4} + \delta_{t,5} + \delta_{s,1} - \delta_{s,2}.$$

Thread releasing `leader` *lock:*

$$\varphi_6(y) = r_l y_{t,5} x_{l,2},\quad e_6(y) = -\delta_{t,5} + \delta_{t,6} + \delta_{l,1} - \delta_{l,2}.$$

Thread accepting a request from client:

$$\varphi_7(y) = r_i \min(x_{c,3}, x_{t,6}),\ \ e_6(y) = -\delta_{c,3}+\delta_{c,4}-\delta_{t,6}+\delta_{t,7},$$

Thread acquiring lock on `epoll`*:*

$$\begin{aligned}\varphi_8(y) &= \frac{1}{y_{t,2}+y_{t,7}} r_l y_{t,7} y_{s,1} H(y_{t,2}+y_{t,7})\\ e_8(y) &= -\delta_{t,7} + \delta_{t,8} - \delta_{s,1} + \delta_{s,2}.\end{aligned}$$

This is modeled similarly to φ_2.

Thread reactivating socket descriptor:

$$\varphi_9(y) = r_a \delta_{t,8} \delta_{s,2},\quad e_9(y) = -\delta_{t,8} + \delta_{t,9}.$$

Thread releasing lock on `epoll`*:*

$$\varphi_{10}(y) = r_l \delta_{t,9} \delta_{s,2},\quad e_{10}(y) = -\delta_{t,9} + \delta_{t,10} + \delta_{s,1} - \delta_{s,2}.$$

Thread performing computation:

$$\varphi_{11}(y) = r_c \delta_{t,10},\quad e_{11}(y) = -\delta_{t,10} + \delta_{t,11}.$$

Thread responding to client:

$$\varphi_{12}(x) = r_o \min(x_{c,4}, x_{t,11}),\ e_{12}(x) = \delta_{c,1}-\delta_{c,4}-\delta_{t,1}+\delta_{t,11}.$$

V. Validation

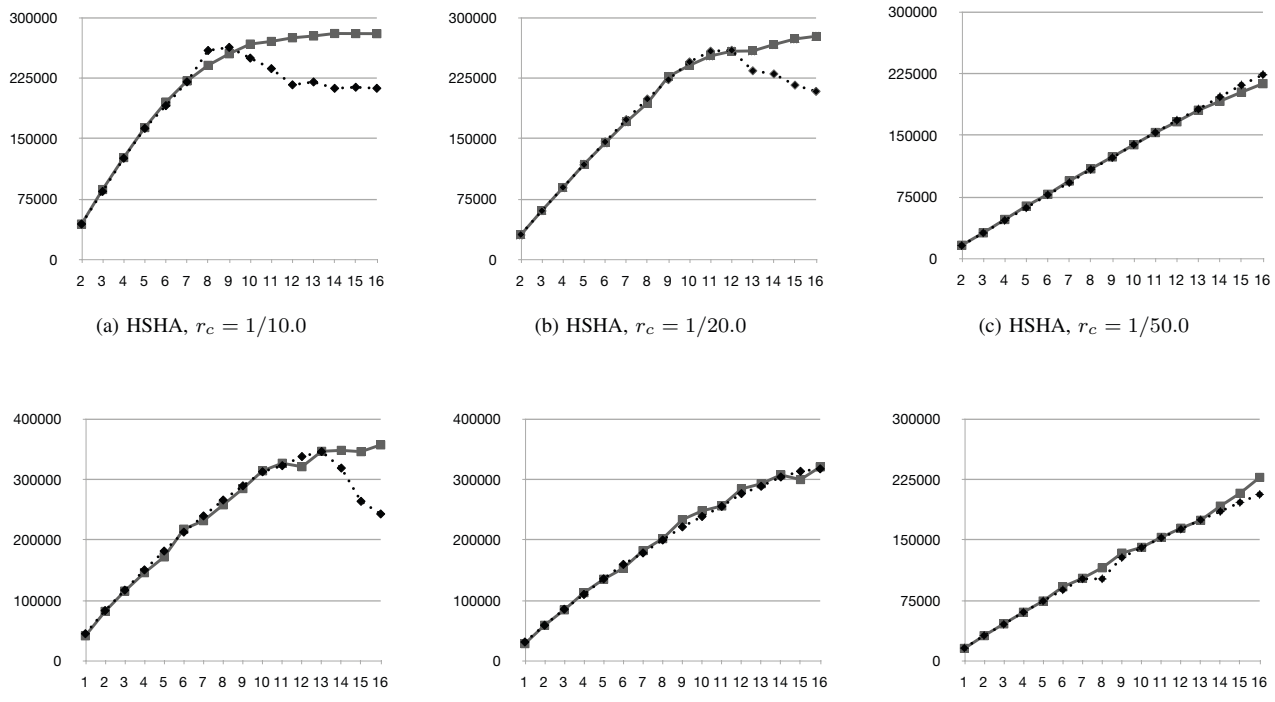
A. Parametrization

The target machine for our validation was a 2x8 AMD Opteron 6134 system, consisting of two processor sockets, each equipped with an octo-core processor. Within each socket are two NUMA (non-uniform memory access) domains such that four processors share L3 cache memory.

The rates $r_a, r_c, r_e, \ldots$ were estimated from measurements conducted in both implementations of HSHA and LF run with the smallest concurrency levels. In HSHA, this amounts to keeping the monitor thread and a single worker, whereas LF used only a single thread (which thus continuously acted as the leader). Any operation performed on locks is characterized by the same rate, r_l, which depends on the implementation used, here a `pthread mutex`. To estimate the actual *service time* of each operation we used LF, where locks were guaranteed not to incur in any queueing delay due to contention. Thus, the rate r_l was set to $1/0.013$ (throughout the remainder of this paper, time units are expressed in μs). Service times for insertion and removal from the shared queue were estimated in a similar manner, and set to $r_p = 1/0.215$ and $r_g = 1/0.256$, respectively. The rate r_t, associated with retrieving a socket descriptor previously returned by an `epoll_wait`, was set to $1/0.01$. The rates for an individual receipt/send of a message were found to be equal to $r_i = 1/9.200$ and $r_o = 1/1.900$, respectively. Implicitly, the structure of the generating functions ϕ_{10}, ϕ_{15}, φ_7, and φ_{12} assume a network configuration where there is enough bandwidth for each connection. A specific network configuration where this does not hold should be reflected in suitable manipulations of the aforementioned generating functions. This aspect is not further considered in the remainder of this paper.

As discussed, the rate for `epoll_wait` was estimated as a function of h, the number of socket descriptors returned, r_e/h; that is, r_e is the rate when only one handle is returned. Two different values were obtained for HSHA and LF, specifically $1/2.900$ and $1/0.635$, respectively. The difference is caused by different usage patterns of LF and HSHA. The monitoring thread of HSHA always waits for new events. If none is available, the `epoll` API puts the thread to sleep but waking it requires additional time. On the other hand, threads of LF find several events per invocation and thus, are never put to sleep. The service rate for reactivation only deals with a single socket descriptor at a time; it was estimated to be $r_a = 1/0.589$.

In this study, the rate of computation, r_c, is an input to the model, in order to study its accuracy across different workload levels. In the implementation, a given value of r_c was realized by means of a synthetic 100% CPU-bound workload by means of busy waiting for a predefined period of time.



(a) HSHA, $r_c = 1/10.0$ (b) HSHA, $r_c = 1/20.0$ (c) HSHA, $r_c = 1/50.0$

(d) LF, $r_c = 1/10.0$ (e) LF, $r_c = 1/20.0$ (f) LF, $r_c = 1/50.0$

Figure 3: Validation results for HSHA and LF for varying r_c. x-axis: threading concurrency levels (N); y-axis: steady-state throughputs; grey solid lines with square markers: model results; black dotted lines with diamond markers: measurements.

B. Set-up and comparison

The system was run long enough to measure the steady-state throughput of messages by dividing the observations into 5 equally sized batches which were found to have 95% confidence intervals below 1%. The so-obtained averages were compared against the results of stochastic simulation of the CTMCs, using the same stopping criteria. In the models, the throughput was computed as the expected value of the function ϕ_{15} for HSHA, and φ_{12} for LF, over the steady-state probability distributions of the chain. For each design pattern, we considered different workloads $r_c = 1/10$, $1/20$, and $1/50$ and different concurrency levels (for HSHA the total number of threads reported in the following also includes the monitor thread). All the other parameters were kept fixed as discussed in the previous section.

Figure 3 reports the overall results for HSHA and LF respectively. The following observations may be made.

(i) Both models show generally good accuracy across all instances considered, consistently capturing the system's peak throughput. We remark that the predictive power of these model is high, as they are parametrized with information (rate estimates) obtained from measuring the system with the lowest possible concurrency level.

(ii) The quality of the approximation increases with increasing workload, i.e., decreasing values of r_c. This is because higher computational workloads represent the system bottleneck, which are separated by some orders of magnitude from the time scales of the activities related to the specific design pattern (and operating system/machine) under consideration.

(iii) The models are also in agreement with respect to the fact that an LF implementation performs better than HSHA for all computation workloads at the same multiplicity level, i.e., when there are N worker threads in LF and $N-1$ service threads and 1 monitor thread in HSHA. This consistent behavior is due to the implementation chosen for the evaluation. HSHA is known to be better suited in cases where incoming requests are subjected to additional manipulation, e.g., prioritization or ordering. The extra cost of such operations can be suitably incorporated in our HSHA performance model.

C. Range of validity

Unlike the model, the measurements show performance degradation after the peak throughput is attained. This is particularly evident for smaller workloads, where the model is however still accurate for concurrency levels below that ones yielding peak throughputs. The results show that degradation is more pronounced for LF—for instance, in the case $r_c = 1/10$, the ratio between the peak throughput (for $N = 13$) and most degraded case was 1.42 for LF and 1.24 for HSHA (here the peak throughput occurs at $N = 9$.

This behavior may be imputed by the memory access

pattern of `epoll_ctl` and the memory architecture. Since data is shared, updating the rb-tree leads to frequent invalidations of copies in remote caches which is followed by data movement when the worker threads have to access the tree elements. With higher concurrency levels this creates considerable contention on the memory architecture. This effect is increased by the fact that in cross NUMA domain remote memory accesses are notoriously more expensive. Neither the memory access pattern nor the memory architecture is explicitly described in the model.

These phenomena have been already observed. The authors of [17] present a simple benchmark that shows that multi-socket systems exhibit degradation for higher concurrency levels whilst single-chip systems do not. In [18], memory access latencies are presented for current multi-socket systems from AMD and Intel. The authors also show that acquiring and releasing a spinlock under contention takes up to 20 times longer than in the case without contention due to cache line sharing and frequent invalidation.

VI. Conclusions

We have presented continuous-time Markov chain models of two frequently used design patterns for parallel message processing, namely Half-Sync/Half-Async and Leader/Followers. We considered a common approach which uses the same level of abstraction that gives a detailed model of the application-level logic, while abstracting from operating system programming interfaces. We showed that these models predict the performance increase and peak performance very well over a range of message-processing workloads.

The functional laws that govern our models are machine independent, therefore they lend themselves well to being applied to other hardware architectures. Furthermore, we stress their high predictive power, yielded by transition rates which can parametrized from measurements on deployments with the smallest concurrency levels. This amounts to needing only two threads in Half-Sync/Half-Async (the monitor thread and a worker) and one in Leader/Followers.

Our validation study showed that performance degradation is not captured by the models. We speculated that this is due to the fact that, currently, memory is not explicitly characterized. We intend to study the nature of this disagreement in more detail, and provide suitable extensions of our models that address this issue.

Acknowledgement: The work of Mirco Tribastone has been partially sponsored by the EU project ASCENS, 257414.

References

[1] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[2] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture: A System of Patterns*. Wiley, 1996.

[3] M. Fowler, *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2003.

[4] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, *Pattern-Oriented Software Architecture Volume 2: Patterns for Concurrent and Networked Objects*. John Wiley & Sons, 2000.

[5] D. C. Schmidt and C. D. Cranor, "Half-Sync/Half-Async — An Architectural Pattern for Efficient and Well-structured Concurrent I/O," in *Proceedings of the 2nd Annual Conference on the Pattern Languages of Programs*. Addison-Wesley, 1996, pp. 1–10.

[6] D. C. Schmidt, C. O'Ryan, M. Kircher, and I. Pyarali, "Leader/followers — a design pattern for efficient multi-threaded event demultiplexing and dispatching," University of Washington, Tech. Rep., 2000.

[7] M. Kerrisk, *The Linux Programming Interface*. No Starch Press, 2010.

[8] R. Strebelow and C. Prehofer, "Analysis of event processing design patterns and their performance dependency on i/o notification mechanisms," in *Int. Conf. MSEPT'12*, in-press.

[9] B. O'Sullivan and J. Tibell, "Scalable I/O event handling for GHC," *SIGPLAN Not.*, vol. 45, no. 11, pp. 103–108, Sep. 2010.

[10] Y. Zhang, C. Gill, and C. Lu, "Real-Time Performance and Middleware for Multiprocessor and Multicore Linux Platforms," in *Proceedings of RTCSA'09*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 437–446.

[11] H. Gomaa and D. A. Menascé, "Performance engineering of component-based distributed software systems," in *Performance Engineering, State of the Art and Current Trends*. London, UK, UK: Springer-Verlag, 2001, pp. 40–55.

[12] D. Petriu, C. Shousha, and A. Jalnapurkar, "Architecture-based performance analysis applied to a telecommunication system," *IEEE Trans. Softw. Eng.*, vol. 26, no. 11, pp. 1049–1065, Nov. 2000.

[13] G. Franks, T. Omari, C. M. Woodside, O. Das, and S. Derisavi, "Enhanced modeling and solution of layered queueing networks," *IEEE Trans. Software Eng.*, vol. 35, no. 2, pp. 148–161, 2009.

[14] N. Mani, D. Petriu, and M. Woodside, "Studying the impact of design patterns on the performance analysis of service oriented architecture," in *37th EUROMICRO SEAA Conference*, 2011, pp. 12 –19.

[15] J. Eckhardt, T. Mühlbauer, M. AlTurki, J. Meseguer, and M. Wirsing, "Stable availability under denial of service attacks through formal patterns," in *15th International Conference on Fundamental Approaches to Software Engineering (FASE'12)*, ser. LNCS. Springer, 2012.

[16] S. Hong and H. Kim, "An integrated GPU power and performance model," *SIGARCH Comput. Archit. News*, vol. 38, no. 3, pp. 280–289, Jun. 2010.

[17] J. Moses, R. Illikkal, L. Zhao, S. Makineni, and D. Newell, "Effects of locking and synchronization on future large scale cmp platforms," in *Proceedings of the 9th Workshop on Computer Architecture Evaluation using Commercial Workloads (CAECW), along with HPCA-12*, 2006.

[18] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang, "Corey: an operating system for many cores," in *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, ser. OSDI'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 43–57.