Generic Messages: Capability-based Shared Memory Parallelism for Event-loop Systems

Luca Salucci
Università della Svizzera italiana (USI), Switzerland
luca.salucci@usi.ch

Daniele Bonetta
Oracle Labs, Austria
daniele.bonetta@oracle.com

Stefan Marr
Johannes Kepler University Linz, Austria
stefan.marr@jku.at

Walter Binder
Università della Svizzera italiana (USI), Switzerland
walter.binder@usi.ch

Abstract

Systems based on event-loops have been popularized by Node.JS, and are becoming a key technology in the domain of cloud computing. Despite their popularity, such systems support only share-nothing parallelism via message passing between parallel entities usually called workers. In this paper, we introduce a novel parallel programming abstraction called Generic Messages (GEMs), which enables shared-memory parallelism for share-nothing event-based systems. A key characteristic of GEMs is that they enable workers to share state by specifying how the state can be accessed once it is shared. We call this aspect of the GEMs model capability-based parallelism.

Categories and Subject Descriptors D.1.3 [*Software*]: Programming Techniques—Concurrent Programming

Keywords Shared memory, event-loop systems, Node.JS, generic messages.

1. Introduction

Programming languages and frameworks based on event-loop programming models have become very popular in domains such as cloud computing and micro-services. Node.JS is the most popular of such frameworks, but several others exist [6], including languages that compile to JavaScript [1, 3].

Many of such languages and frameworks only support a limited model of parallelism, in the form of message passing between share-nothing parallel entities. This model is known as the communicating event-loops model [7]. In this parallel programming model, each event loop runtime (or simply each worker) is replicated as an independent and isolated parallel entity. Workers do not share any memory space, and interact only by means of explicit asynchronous message passing. The model is inspired by actors [4], and is very convenient for programmers, because they do not need to deal with issues such as data races and deadlocks.

Despite its many benefits, share-nothing message-based parallelism is not a one-size-fits-all solution, and approaches based on shared-memory parallelism could give considerable performance benefits, or can be more convenient for expressing certain computations. Nevertheless, shared-memory parallel programming is complex, as it requires developers to deal with synchronization and race conditions. This is particularly true for systems featuring an event-loop, where concurrency is already present in the form of asynchronous I/O. Ideally, any form of shared-memory parallel programming abstraction for such systems should be safe.

In this paper we introduce the notion of *Generic messages* (GEMs), a programming abstraction that can be used to expose safe shared-memory parallel programming models to systems relying on communicating event-loops. At the very high level, a GEM is some form of state (e.g., an object instance) that initially belongs to a single worker (called the GEM owner), and that can be shared via message passing with other workers. Unlike other forms of data sharing, when a GEM is sent to other workers, the state it encapsulates becomes accessible by them in a controlled way. Specifically, the owner of the GEM specifies *how* the GEM can be used by the receiving workers. Therefore, the owner of the shared state specifies upfront the operations that workers are allowed to perform on the state they share. We call this model *capability-based* parallelism[1], meaning that at the moment a GEM is shared, its owner not only exposes its state, but also limits the provided capabilities to interact with it. An intuitive example for a capability is read-only access. Beyond that, GEMs enable many advanced and fine-grained policies, e.g., partitioned access or temporal immutability. Rather than enabling a specific parallel programming model, generic messages are fully customizable, and can expose shared-memory to workers in several ways.

2. GEMs and Capability-based Parallelism

A GEM is a type of object that can be exchanged between workers via message-based interactions to enable shared-memory parallel programming. It can be seen as a capability granting controlled access to shared-memory, and can be considered a combination of the following elements:

(1) A *shared object*, that is, an object graph to be shared with multiple workers through the GEM.
(2) *Dynamic sharing semantics* that controls how the shared object can be accessed in parallel by multiple workers.

Shared objects can be associated only with one GEM at a time, i.e., it is not possible to reference the same object instance from two different GEMs, neither directly nor indirectly. Different GEMs can encapsulate different sharing semantics, and therefore enable multiple parallel programming models.

At its core, a GEM is an object with the following components:

- **GEM public API**: a GEM-specific API that is accessible to all the workers receiving the GEM in the form of a message.

[1] The model is inspired by the capability system present in the Unix OS. Unlike Unix capabilities, GEMs can be used to develop safe parallel applications.

PPoPP '16 March 12-16, 2016, Barcelona, Spain

DOI: http://dx.doi.org/10.1145/2851141.2851184

Master

```
1  // Multicast the GEMs to the workers
2  workers.multicast(inputFile, keywords)
3     .gather(function(result) {
4        console.log('Total matching lines are:' + result);
5     });
```

Workers

```
1  // GEMs received: 'lines' is partitioned, 'keywords' is read-only
2  worker.on('message', function(lines, keywords) {
3     var result = {};
4     // Access both read-only and partitioned gems
5     // The partitioned GEM has the 'getRange' API
6     // used to retrieve partitions dynamically
7     lines.getRange(function(from,to) {
8        for (var l = from; l < to; l++)
9           for (var key in keywords)
10             if (lines[l].split().contains(key))
11                count(key,result);
12       // Both GEMs enforce strict access control
13       keywords[42] = 42; // read-only GEM: throws an exception!
14       lines[to+1] = 42; // out-of-bound: throws an exception!
15    });
16    // A message with the result can be sent back to the master
17    worker.reply(result);
18 });
```

Figure 1: A Node.JS text scraping application using two GEMs.

- **GEM meta API**: a custom meta-object protocol API [5] that is used by all the workers using the GEM.
- **GEM state**: global state accessible to all GEMs in all workers, and local state private to the worker that receives a GEM. State is private, and can be accessed only by the GEM public and meta API.

By combining these three components, GEMs can specify any custom sharing semantics. Examples of GEMs with different sharing capabilities are:

(1) **ReadOnly GEM**: Every worker has read-only access to all elements of the shared object. Attempts to write to any of the elements cause an exception.

(2) **Owned GEM**: Only one worker at a time has exclusive access to all elements of the shared object. Attempts to perform concurrent reads or writes cause an exception on the worker that does not have the exclusive read/write access.

(3) **Partitioned GEM**: Workers have read and write access to disjoint subsets of elements of the shared object. Attempts to read or write outside of the partition cause an exception. Depending on the GEM implementation, the partition can be assigned statically or dynamically.

Other, more advanced GEMs exist that can support more complex forms of access to shared-memory, for example enabling safe concurrent access to the shared object.

Figure 1 shows a simple Node.JS application using two of the GEMs described above. It does text scraping on an input file owned by a master worker; the file is scanned line-by-line by multiple workers that compare the text content against a dictionary of keywords. The two GEMs used in the example are a read-only GEM and a partitioned GEM. The former is used to grant read-only access to the dictionary of keywords to all the workers. In a share-nothing scenario, the dictionary would have to be replicated among all the workers, while the GEM ensures concurrent read-only access and prevents concurrent writes. The latter GEM is used to partition the input file to be scraped among workers, to achieve data parallelism. Partitioning is handled dynamically by the GEM, which exposes a simple API (i.e., *getRange*) that can be used by workers to retrieve a slice of the shared object. Without shared-memory parallelism, each partition of the input file would have to be copied into the memory space of the receiving worker. Using the GEM, no copies are required: once it is received, each worker becomes the exclusive owner of a partition, and gains exclusive read and write access over it. The GEM guarantees safety by ensuring that workers do not access an index of the array that they do not own (see line 14 in Figure 1). Message passing can still be used with GEMs. As shown in the example (line 17), messages can be used to communicate final or intermediate results once they are available.

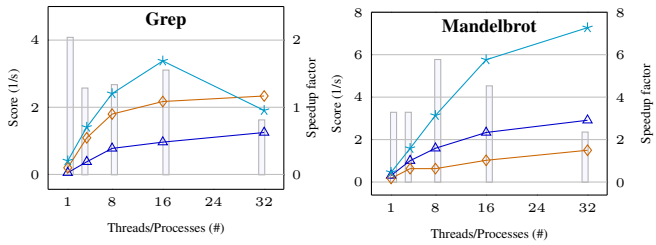

Figure 2: GEMs performance of two Node.JS applications. For each benchmark the reported score is the inverse of the execution time ($1/t$). The GEM-based implementation of the benchmarks (—✶—) outperforms the message-based implementations in Node.JS (—△—), and Graal.JS (—◇—). The speedup factor of GEMs over Node.JS is also reported ( ).

3. Evaluation

The GEMs model is a generic extension to communicating event-loop frameworks or languages. We developed an initial version of the GEMs model in the context of Node.JS, building upon the GraalJS [2] JavaScript engine, a fully-compliant ECMA6 implementation of Node.JS for the JVM. As an initial evaluation, we ported message-based benchmarks for Node.JS to the GEM model. The results are depicted in Figure 2. The experiments ran on a server-class machine using Ubuntu 12.04, equipped with 128 GB of RAM and two 8-core Intel(R) Xeon(R) CPU E5-2680 (2.70 GHz). As the figure shows, the GEM-based implementations offer better performance over both the Graal.JS and Node.JS implementations that use message passing. The reason for the speedup is the efficient usage of shared memory, and the reduced communication overhead between workers.

Acknowledgments

The research presented here has been supported by Oracle Labs (ERO project 1332) and by the Swiss National Science Foundation (project 200021_153560). Stefan Marr has been supported by Oracle Labs. We thank all members of the Virtual Machines Research Group at Oracle Labs for their help and support.

References

[1] Dart: Scalable, productive app development. `https://www.dartlang.org/`.

[2] Oracle Graal.JS, High-Performance JavaScript on the JVM. `http://www.oracle.com/technetwork/oracle-labs`.

[3] Scala.js: A safer way to build robust front-end Web applications! `http://www.scala-js.org/`.

[4] G. Agha. *Actors: a model of concurrent computation in distributed systems*. MIT Press, Cambridge, MA, USA, 1986.

[5] G. Kiczales, J. des Rivires, and D. G. Bobrow. *The Art of the Metaobject Protocol*. The MIT Press, Cambridge, 1991.

[6] S. Marr and H. Mössenböck. Optimizing Communicating Event-Loop Languages with Truffle, October 2015. Presentation at 5th AGERE Workshop, colocated with SPLASH'15.

[7] M. S. Miller, E. D. Tribble, and J. Shapiro. Concurrency Among Strangers: Programming in E as Plan Coordination. In *Symposium on Trustworthy Global Computing*, volume 3705 of *Lecture Notes in Computer Science*, pages 195–229. Springer, April 2005.