Efficient Leader Election for Synchronous Shared-Memory Systems

Vicent Sanz Marco, Michael Zolda, Raimund Kirner
University of Hertfordshire, Hatfield, UK

Abstract—Leader election is a frequent problem for systems where it is important to coordinate activities of a group of actors. It has been extensively studied in the context of networked systems. But with the raise of many-core computer architectures, it also became important for shared-memory systems.

In this paper we present an efficient leader election technique for synchronous shared-memory systems. Synchronous in our context means the response time of code sections with relevant communication patterns is bounded. Our leader election method is used to help making the scheduling layer LPEL fault tolerant. With our approach LPEL will be efficient to resolve problems when the leader fails.

Keywords—fault tolerance, leader election, parallel computation, shared memory, synchronous systems

1 INTRODUCTION

Fault tolerance is the ability of a system to process information, even in the presence of a failure or anomaly in the system. As part of our work to improve fault-tolerance in LPEL [8], we need a protocol for leader re-election, which is called conductor in LPEL terminology [6].

Master election is a well-studied problem in computer science, especially for distributed systems [9], [2].

This paper presents a method to elect a leader in a synchronous shared-memory system. Our aim is to create a fast method with a small memory footprint that only involves nodes that are not busy. Busy nodes do not participate in the election, and instead learn about the new leader later.

2 LPEL

The Light-Weight Execution Layer (LPEL) was developed to provide an efficient and flexible execution platform for application based on stream processing on architectures with shared memory [8], [6]. A stream-based coordination program is a set of components connected by channels called streams.

LPEL provides efficient scheduling and communication mechanisms for stream-based computing. Central features of LPEL are lightweight user-level task scheduling and lightweight synchronization techniques to keep the overhead at a minimum. The efficiency of the communication mechanisms is due to user-level scheduling, for which it is known to LPEL that the data structures can be really accessed in parallel from different cores, and which access of data structures is serialized due to access from within tasks running on the same core.

With the architecture of LPEL, it is possible to deliberate the allocation of available processing resources. Also, LPEL allows handling a large number of tasks at the same time with lock-free synchronization techniques and user-level threading.

LPEL provides good scalability and makes use of multiple available processor cores. The scalability is ensured because LPEL is using data structures and lock-free techniques. Further characteristics of LPEL are:

• Support for non-determinism by testing the availability of new data on input chan-
nels.
- Dynamic (de-)construction of the streaming network during runtime.
- Provide the possibility to adapt the scheduling policy to the needs of the application.

LPEL is implemented upon the scheduling model of Parks [7] for process networks: The connection of the tasks are uni-directional and are by streams, which are modeled as bounded buffers. The tasks are suspended from execution upon reading from a full stream and writing to an empty stream. This model allows for an easy implementation and lends to parallel execution. Bounded buffers have the advantage of being able to provide back-pressure, but the downside is that it can lead to artificial deadlocks in circular networks. The tasks are not directly executed as operating system threads, but they are executed as user-level threads in the context of LPEL.

3 Related Work

The masters election, also called leader election, is a known problem in computer science. There exists a significant amount of research results related to this problem. Leader election is about the problem of choosing a leader or coordinator among a set of communicating nodes. Related work on leader election can be classified of whether it focuses on synchronous or asynchronous systems, and whether it focuses on distributed/networked or shared-memory systems.

The problem has been discussed extensively for distributed systems. For example, the book written by Hromkovic [4] presents different algorithms of leader election in asynchronous distributed networks. Ingram et al. studied leader election for asynchronous network with dynamically changing communication topology [5]. In contrast to leader election in networked systems, we focus on systems with shared memory, which can provide a richer choice of communication patterns, e.g., atomic test-and-set operations in hardware to efficiently realise synchronisation mechanisms like semaphores.

Guerraou and Raynal proposed a protocol for leader election on asynchronous shared-memory systems [3]. Dharmadeep and K. Gopinath described fault-tolerant algorithms for the leader election in an asynchronous shared memory system [1].

Though our paper also focuses on shared memory systems, we focus at the same time on synchronous systems instead of asynchronous systems. This allows us to use a more efficient approach than those presented for distributed/networked and/or asynchronous systems. At the same time our required notion of synchronism is a rather weak one, as we only require to know time bounds of the code of a few synchronisation patterns. To know the time bounds of such code patterns for embedded systems is realistic, as it consists only of the worst-case execution time (WCET) plus any further scheduling-related delays. With simple code structures and a suitable time-predictable scheduling method this can be justified. In contrast, many classical application contexts of leader election cannot make that assumption, thus requiring more costly solutions.

4 Design

In LPEL, workers represent parallel execution threads. LPEL uses one worker for each core. One of this workers has the leader role and it is called conductor. The conductor is a special worker that distributes tasks to other workers. There is currently no solution if the conductor stops working. To resolve this problem, we present a method to detect conductor failures and select another worker to take the role of the conductor.

In the beginning the system select the core 0 for the conductor. There is shared memory between all workers that can be read simultaneously by the workers, and only one worker can write at a given time. To control the write access, a semaphore is used. The shared memory contains the id of the current conductor and a global counter for previous conductors.

Each worker also has a local counter. Conductor failures are detected using a timeout: When a worker finishes its work, it sends the task back to the conductor. The conductor does
not send a reply message. Then the worker waits for the timeout. If the reply does not arrive, then the worker will resend the task. If the reply does not arrive again, then the worker will start the method to change the leader.

Figure 1 shows a system with two workers and one conductor. The conductor sends $T_1$ to the worker $W_1$ and $T_2$ to worker $W_2$. When the workers are working, the conductor stops to work. $W_1$ finishes $T_1$ and $W_1$ tries to give back $T_1$ to the conductor. $W_1$ waits for a certain time the conductor reply of $T_1$. The timeout of $W_1$ finishes. Then $W_1$ will try to send again $T_1$. If this second chance fails, $W_1$ will start the election protocol. In this moment, $W_2$ finishes $T_2$. First, $W_1$ will try to lock the semaphore. If $W_1$ locks the semaphore, $W_1$ will compare its local counter to the counter in shared memory. If they are the same, $W_1$ will try to write to the shared memory. If they are different, $W_1$ will unlock the semaphore and it will change its local counter and it will use the new conductor. At the same time, $W_2$ tries a second time to send $T_2$ to the worker. In this example, $W_1$ locks the semaphore and $W_1$ writes itself in the shared memory. $W_1$ will be the new conductor. $W_1$ start the initiation of the entire task to do this role. When $W_1$ finishes all the preparation, $W_1$ unlocks the semaphore. $W_2$ tries to write in the shared memory, but $W_2$ cannot write because $W_1$ lock the semaphore. Then $W_2$ waits a certain time to try again to start the election protocol.

The timeout of $W_2$ finishes, then $W_2$ will try to lock the semaphore. $W_2$ will lock the semaphore and it will compare its local counter with the counter in shared memory. They are different, then $W_2$ will unlock the semaphore and it will update its local counter. Maybe with the time core 0 comes back to work, and then it will come back like a normal worker.

4.1 Algorithm

Algorithm 1 shows the steps follows for the workers:

The first and second lines are executed for the entire workers to read the share counter and they will compare with the local counter. This conditional will separate the nodes involved in the election protocol. If the local counter is different, the node was working while a new master was elected, then it only needs to update its local information.

If the local counter and the shared counter are equals, then the node is involved with the election protocol. In this case, the node will try to lock the semaphore. After the node will lock the semaphore, it will compare the local counter and shared counter. If the local counter and shared counter are different, means there is a new leader in the system and the node do not need to be the new leader. If the local counter and shared counter are the same, then this node will be the new leader and it will take the new role. After it will unlock the semaphore and the other nodes can take it.

With this algorithm, it is possible to calculate

$\text{Algorithm 1}$ Algorithm for election protocol

$\begin{align*}
1: & \ g \leftarrow \text{readSharedCounter} \\
2: & \text{if} \ localCounter \neq g \text{ then} \\
3: & \hspace{1em} \text{lock} \leftarrow \text{getSemaphore} \\
4: & \hspace{1em} g \leftarrow \text{readSharedCounter} \\
5: & \text{if} \ localCounter \neq g \text{ then} \\
6: & \hspace{1em} \text{becomeLeader} \\
7: & \hspace{2em} \text{sharedCounter} \leftarrow \text{sharedCounter} + 1 \\
8: & \hspace{2em} \text{setGlobalId} \\
9: & \hspace{2em} \text{releaseLock} \\
10: & \text{else} \\
11: & \text{releaseLock} \\
12: & \text{updateLocalCounter} \\
13: & \text{end if} \\
14: & \text{else} \\
15: & \text{updateLocalCounter} \\
16: & \text{end if}
\end{align*}$
the expected computation time used to complete this election protocol.

Algorithm 2 Algorithm for election protocol

\[
g \leftarrow \text{readSharedCounter}
\]
2: \textbf{if} localCounter \neq g \textbf{then} \{ \begin{align*}
& \text{compareCounter} \allowbreak \\
& \text{getSemaphore} \allowbreak \\
& \text{readSharedCounter} \allowbreak \\
& \text{compareCounter} \allowbreak \\
& \text{body} \allowbreak \\
& \text{releaseLock} \allowbreak \\
& \text{updateLocalCounter} \allowbreak \\
& \text{releaseLock} \text{ and } t_{\text{releaseLock}} \allowbreak \\
& t_{\text{localUpdate}} \allowbreak \\
& \text{end if} \allowbreak \\
\end{align*}
\]
else
\{ \begin{align*}
& \text{releaseLock} \allowbreak \\
& \text{updateLocalCounter} \allowbreak \\
& t_{\text{localUpdate}} \allowbreak \\
& \text{end if} \allowbreak \\
\end{align*}
\]

Algorithm 2 separates the Algorithm 1 inblocks with different labels to calculate the time. There are the next different labels: \(t_{\text{compareCounter}}, t_{\text{getSemaphore}}, t_{\text{body}}, t_{\text{releaseLock}}\) and \(t_{\text{localUpdate}}\). With this hypothesis, in the worst case all the nodes will do the block \(t_{\text{compareCounter}}\). After this, only one node will do the sections: \(t_{\text{getSpinLock}}, t_{\text{compareCounter}}, t_{\text{body}}\) \(t_{\text{releaseLoc}}\). All the others, called master applicants, will do: \(t_{\text{getSpinLock}}, t_{\text{compareCounter}}\) and \(t_{\text{releaseLock}}\).

If in the system there is \(n\) nodes and there is a variable, called \(K\), that it is all the extra time that the master applicants are trying to get the semaphore. The resulting formulae are:

- All the nodes \(\rightarrow n^* t_{\text{compareCounter}}\)
- Winner \(\rightarrow t_{\text{getSpinLock}} + t_{\text{compareCounter}} + t_{\text{body}} + t_{\text{releaseLock}}\)
- Master applicants \(\rightarrow K(n-1)^* t_{\text{getSpinLock}} + (n-1) t_{\text{compareCounter}} + (n-1) t_{\text{releaseLock}}\)

For the calculation of the time, it is necessary to substitute the variable \(K\). This variable is related with the time used for the winner node:

\[
t_{\text{winner}} \leftarrow K * t_{\text{getSpinLock}} \\
K \leftarrow t_{\text{winner}} / t_{\text{getSpinLock}}
\]

If we substitute in the master applicants formula and sum together all the times, the solution of the formula is:

\[
(3n-1)^* t_{\text{compareCounter}} + (n)^* t_{\text{getSpinLock}} + (n)^* t_{\text{body}} + (2n -1)^* t_{\text{releaseLock}}
\]

In conclusion, this election election protocol has a timing complexity of \(O(3n)\).

5 Conclusion

Leader election algorithms play a vital role in distributed environments. In this paper is presented a new method for leader election with shared memory. This method involves only the nodes available having currently a demand for the services from the leader. Other nodes which currently do not need services from the leader, will just approve the decision being made once they demand for a leader later. The protocol has a small execution time and the required space of shared memory is tiny and constant, regardless of the number of nodes. Our use case of this leader election is to make the tasking layer LPEL tolerant against certain faults. As future work, the leader election method will be implemented and tested in LPEL.

References