TALENT: Dynamic Platform Heterogeneity for Cyber Survivability of Mission Critical Applications*

Hamed Okhravi, Eric I. Robinson, Stephen Yannalfo,
Peter W. Michaleas, and Joshua Haines
MIT Lincoln Laboratory
Massachusetts Institute of Technology
Lexington, MA
Email: {hamed.okhravi, erobinson, stephen.yannalfo,
pmichaleas, jhaines}@ll.mit.edu

Adam Comella
Computer Science Department
Rensselaer Polytechnic Institute
Troy, NY
Email: comela@rpi.edu

Abstract—**Despite the significant amount of effort that often goes into securing mission critical systems, many remain vulnerable to advanced, targeted cyber attacks. In this work, we design and implement TALENT (Trusted dynAmic Logical hEterogeNeity sysTem), a framework to live-migrate mission critical applications across heterogeneous platforms. TALENT enables us to change the hardware and operating system on top of which a sensitive application is running, thus providing cyber survivability through platform diversity. Using containers (a.k.a. operating system-level virtualization) and a portable checkpoint compiler, TALENT creates a virtual execution environment and migrates a running application across different platforms while preserving the state of the application. The state, here, refers to the execution state of the process as well as its open files and sockets. TALENT is designed to support a general C application. By changing the platform on-the-fly, TALENT creates a moving target against cyber attacks and significantly raises the bar for a successful attack against a critical application. Our measurements show that a full migration can be completed in about one second.**

I. INTRODUCTION

Mission critical systems are an integral part of national cyber infrastructure. Critical infrastructure systems consisting of smart power grid and utilities, air and space control systems, defense infrastructure, and intelligence systems are examples of mission critical systems. Despite the significant amount of effort and resources used to secure these systems, many remain vulnerable to advanced, targeted cyber attacks. Complex systems and commercial-of-the-shelf (COTS) components often exacerbate the problem.

In this work, we design and implement TALENT (Trusted dynAmic Logical hEterogeNeity sysTem), a framework to live-migrate mission critical applications across heterogeneous platforms. We hypothesize that in mission critical systems the mission itself is the top priority, not individual instances of the component. By live-migrating the mission critical application from one platform to another, TALENT can thwart cyber attacks. Also by dynamically changing the platform at randomly chosen time intervals, TALENT creates a moving target against attackers. This means the information collected by the attacker about the platform during the reconnaissance phase becomes ineffective at the time of attack.

TALENT has several design goals:

- Heterogeneity at the instruction set architecture (ISA) level, which means we should be able to run the application on top of processors with different instruction sets.
- Heterogeneity at the operating system level.
- Preservation of the *state* of the application including open files, sockets, and execution state of the process. This is an important property in mission critical systems because simply restarting the application from scratch on a different platform may have undesired consequences.
- Working with a general purpose, system language (such as C). Many of the functionalities in TALENT are straight forward to implement in a platform independent language like Java because the Java virtual machine (JVM) provides a sandbox for the application. However, many commodity and COTS systems are developed in C. Restricting the system to only Java-like languages limits its usage.

TALENT must provide OS and hardware heterogeneity while preserving the state and environment despite the binary incompatibility of different architectures. TALENT addresses these challenges with two main

* This work is sponsored by the Department of Defense under Air Force Contract FA8721-05-C-0002. Opinions, interpretations, conclusions, and recommendations are those of the authors and are not necessarily endorsed by the United States Government.

ideas:

- OS-level virtualization (a.k.a container-based OS) to sandbox the application and migrate the environment including the filesystem, open files, and network sockets.
- Portable checkpoint compilation to compile the application for different architectures and migrate the execution state across different platforms.

The rest of the paper is organized as follows. Section II describes the threat model used throughout the paper. Section III explains the design of TALENT and its main features. The implementation of TALENT is described in Section IV. Section V provides the initial evaluation results on the performance of TALENT. We review the related work in Section VI before concluding the paper in Section VII.

II. Threat Model

The threat model in TALENT assumes there is an external adversary trying to exploit a vulnerability in the system (either in the OS or the binary of the application) in order to disrupt the normal operation of a mission critical application. For simplicity and on-the-fly platform generation, we use a hypervisor (hardware-level virtualization). The threat model assumes the hypervisor and the hardware of the system are trusted. Hardware-based cryptographic verification (e.g. using TPM) will check the authenticity of the hypervisor, but we further assume the implementation of the hypervisor is bug free. The OS-level virtualization logic must also be trusted. The remaining system (including the OS and the applications), however, is not trusted and may contain vulnerabilities or malicious logic.

It is also assumed that although an attack is feasible against a number of different platforms (OS/architecture combinations), there exist a platform against which the attack is not applicable. This means not *all* the platforms are vulnerable. The goal in TALENT is to migrate the mission critical application to a different platform at random time intervals when a new vulnerability is discovered or when an attack is detected. Attack detection can be done using various techniques described in the literature [4], but it is beyond the scope of this paper.

Heterogeneity at different levels can mitigate various attacks. Application-level heterogeneity mitigates binary and architecture specific exploits and malicious compiler. OS-level heterogeneity mitigates kernel specific attacks, OS specific malwares, and OS persistent attacks (rootkits). Finally, hardware heterogeneity can thwart supply chain attacks, malicious/faulty hardware, and architecture specific attacks. TALENT is not a complete defense against all these attacks; it can, however, provide survivability in the presence of platform specific attacks by means of dynamic heterogeneity.

III. Design

To address the challenge involved in using heterogeneous platforms including binary incompatibility and loss of state and environment, TALENT uses two key ideas: OS-level virtualization and portable checkpoint compilation.

A. OS-level virtualization and Environment Migration

An important goal of the project is to preserve the environment of a mission critical application. The environment includes the filesystem, configuration files, open files, network connections, and open devices. Note that many of the environment parameters can be preserved using VM migration. However, VM migration can only be done with a homogeneous OS and hardware. Because we want to change the OS and hardware while migrating a live application, VM migration is not applicable. TALENT uses OS-level virtualization to sandbox an application and migrate the environment. Here we provide a quick overview of OS-level virtualization.

1) OS-level virtualization: OS-level virtualization is a method where a kernel allows for multiple isolated user-level instances. Each instance is called a container (jail or virtual environment).The method was originally designed to support fair resource sharing, load balancing, and cluster computing application. OS-level virtualization can be thought of as extended `chroot` in which all resources (devices, filesystem, memory, sockets, etc.) are virtualized.

Note that the major difference between OS-level virtualization and hardware-level (e.g. Xen and KVM) is the semantic level at which the entities are virtualized (Figure 1). Hardware-level hypervisors virtualize disk blocks, memory pages, hardware devices, and CPU cycles, whereas OS-level virtualization works at the level of filesystems, memory regions, sockets, and kernel objects (e.g. IPC memory segments and network buffers.) Hence, the semantic information often lost in hardware virtualization is readily available in OS-level virtualization. This makes OS-level virtualization a good choice for use cases where this information is needed such as monitoring or sandboxing at the application level.

2) Environment Migration: TALENT uses OS-level virtualization to migrate the environment. When a migration is requested (as a result of a malicious activity or simply a periodic migration), TALENT migrates the container of the application from the source machine to the destination machine. This is done by synchronizing the filesystem of the destination container with the source container. Since the OS keeps track of open files, the same files are opened in the destination. Because this information is not available at the hardware virtualization level (the semantic gap between files and disk blocks), additional techniques are required to recover it (e.g.

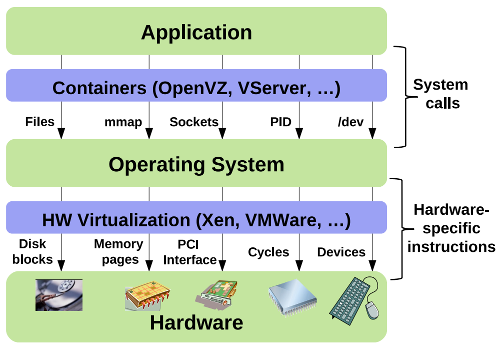

Fig. 1. OS-level and hardware-level virtualization approaches

virtual machine introspection). On the other hand, the information is readily available in TALENT.

Network can be virtualized in a number of ways: second layer, third layer, and socket virtualization. Virtualizing network at the second layer means that each container has its own IP address, routing table, and loopback interface. Third layer virtualization implies each container can listen to any IP address/port and sockets are isolated using the namespace. Finally, socket virtualization means each container can access any IP address/port and sockets are isolated using filtration. In TALENT we use the first approach. To preserve network connections during migration, the IP address of the container's virtual network interface is migrated to the new container. Then the state of each TCP socket (kernel's `sk_buff`) is transferred to the destination. The network migration is seamless to the application, and the application can continue sending and receiving packets on its sockets.

Many OS-level virtualizations also support IPC and signal migration. In each case, the states of IPC and signals are extracted from the kernel data structures and migrated to the destination. These features are supported in TALENT through OpenVZ [10].

B. Checkpointing and Process Migration

Migrating the environment is only one step in backing up the system because the state of running programs must also be migrated. To do this, a method to checkpoint running applications must be implemented. Once all checkpointed program states are saved in checkpoint files, the state can be migrated by simply mirroring the file system. Checkpointing in TALENT must meet certain requirements.

- *Portability:* Checkpointed programs should be able to move back and forth between different architectures and operating systems in a heterogeneous computing environment.
- *Transparency:* Heavy code modification should not be required to existing programs in order to introduce proper checkpointing.
- *Scalability:* Checkpointed programs may be complex and handle large amounts of data. Checkpointing should be able to handle them without affecting the performance of the underlying system.

To meet the requirement of portability, a portable checkpoint compiler (PCC) is preferable. Figure 2 illustrates a portable checkpoint compilation process. This allows compilation to occur independently on various operating system/architecture pairs. The resulting executable program, including the inserted checkpointing code, functions properly on each platform on which it was compiled.

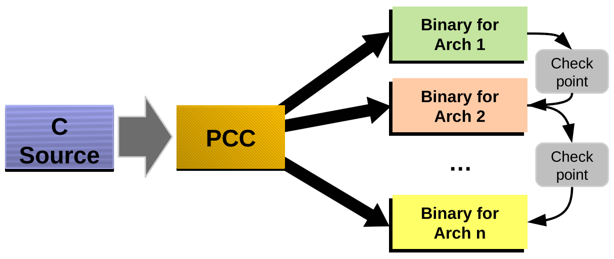

Fig. 2. Portable checkpoint compilation

Transparency is obtained by performing automatic code analysis and checkpoint insertion. This prevents the end user from having to modify their code to indicate where checkpointing should be performed or what data specifically should be checkpointed.

Finally, scalabiliy is obtained in two ways. First, the frequency of checkpointing bottlenecks in the checkpointing process can be controlled. Second, through the use of compressed checkpoint file formats, the checkpoints themselves remain as small as possible even as the amount of data processed by the program increases.

Given these constraints, TALENT has selected the *Controller/Precompiler for Portable Checkpointing* (CPPC) [14] compiler to save the state of a running program. CPPC is capable of storing the program state of a running program in a format that is both operating system and hardware independent, and then correctly restarting that program on a different platform using the state that was previously stored.

IV. Implementation

TALENT is implemented using OpenVZ containers-based OS and CPPC portable checkpoint compiler.

A. Environment Migration

There are different implementations of OS-level virtualization available including OpenVZ [10], LXC [2], and VServer [18] for Linux, Virtuozzo [1] for Windows, and Jail [15] for FreeBSD. We have chosen UNIX-like operating systems as our platform. We have also chosen OpenVZ as the container because of its ease of use, stable code, and support for second layer network virtualization.

We use OpenVZ version 2.6.27 and patch it into different kernels. We use KVM [7] as the underlying hypervisor. We have implemented and tested TALENT on top of Intel Xeon 32-bit, Intel Core 2 Quad 64-bit, and AMD Opteron 64-bit processors. We have tested Gentoo, Fedora (9, 10, 11, and 12), CentOS (4 and 5), Debian (4, and 5), Ubuntu (8 and 9), and SUSE (10 and 11) operating systems. In total we have tested 37 combinations.

For environment migration, we do not use the OpenVZ live migration feature because it migrates the processes within the container causing binary incompatibility. Instead, we migrate the environment by freezing the container, synchronizing the filesystem, migrating the virtual network interface, and transferring buffers, IPC, and signals. We then substitute the binary of the application built for the destination processor. Checkpointing and preserving the application state are explained in detail later in this paper.

B. *Overview of CPPC*

CPPC is a compiler-assisted checkpointing program. It involves four phases during the execution as follows:

1) *Compiling the Code.* The code is compiled on each platform independently.
2) *Configuring the Run.* The preferences for the checkpointing during a run are configured.
3) *Checkpointing.* The run is started, and checkpointing of the state occurs automatically.
4) *Restarting the Run.* The checkpoint (after being migrated) is resumed on a new platform.

This section describes how each of these phases performs its task.

1) Compiling the Code: CPPC is capable of compiling traditional C and Fortran 77 code. It compiles unmodified source code, and the programmer does not need to have any knowledge of checkpointing. CPPC automatically determines how/where to checkpoint a program. An example of how this is done is shown in Section IV-C.

CPPC interfaces with the user code as a precompiler. It uses the Cetus compiler infrastructure [11] to determine the semantic behavior of a program in order to decide where to place checkpointing directives. Once this is determined, the code is re-factored with CPPC directives. The re-factored code can then be compiled using a traditional compiler such as *cc* or *gcc*.

2) Configuring the Run: In order to run with checkpointing, CPPC requires a configuration file. This file specifies parameters used for checkpointing, including the frequency of checkpointing, the number of checkpoints to store, and the location in which to store them. While a default file is provided, typically a user may want to configure this file based on the behavior they expect of their program. For example, the frequency of checkpointing can be modified for programs that checkpoint frequently. This allows a program to avoid file writing bottlenecks.

Changing run parameters can be done directly in the configuration file by modifying the appropriate value, or can be done directly when starting a run using command line options. Typically, a program will have a suitable configuration specified in the configuration file. However, a user may override this if they want to see a different behavior by entering a new value for a parameter on the command line. The configuration file can be stored in either a straight text format or specified using XML via the Xerces package [13].

3) Checkpoint: Checkpoints are stored in a file using an HDF5 format [6]. HDF5 is a compressed file format for storing and managing data. Since this format is widely used and deployed on many platforms, it allows CPPC to store checkpoint files that are compatible across a range of architectures and operating systems. Additionally, a CRC-32-based algorithm is supported to ensure checkpoint file correctness.

As stated previously, checkpointing is done automatically. The user can change the rate at which this is performed through the configuration file/command line options. The compiler options also allows the programmers to manually specify where checkpointing should occur by adding pragmas into their source code. Pragma options also exist for other CPPC functionality such as indicating code which should be run upon restart for re-initialization of data not stored in memory or other initialization tasks like restarting MPI.

4) Restarting from a Checkpoint: After a run has been started and a checkpoint has been recorded, it is possible to restart the run from the last recorded checkpoint. This can be done either on the same platform or on a different platform.

CPPC performs restarts by adding "jump" statements in the original code to the locations of checkpoints. Based on the checkpoint file, it knows which of these locations to jump to upon restart. These jump states are ignored during the initial run, leading CPPC to execute the program as if no changes had been added. In addition to jumping to the appropriate starting location, the checkpoint file contains information about variable values within the program. These are loaded upon restart so that the program resumes in the same state it was in upon checkpointing.

C. Code Example

In order to illustrate how the CPPC checkpointer operates, a simple example of computing a factorial is presented. This example is meant for the purpose of exposition only. CPPC is capable of handling much more complicated code bases than the one presented here.

1) Initial Code: As an example, we take a simple C program to compute the factorial of a value. The code is shown in Figure 3. For simplicity, we choose to have a main function with no inputs.

```
int main(int argc, char **argv)
{
   int fact;
   double curr;
   int i;

   fact = 20;
   curr = 1;
   for(i=1; i<=fact; i++)
   {
      curr = curr * i;
   }
   printf("%d factorial is %f",
          fact, curr);
   return 0;
}
```

Fig. 3. A simple program to compute the factorial of 20

2) Markup Code: CPPC converts this code to a markup code using pragmas to specify where special CPPC content should be inserted. For example, the markup for the code in Figure 3 is shown in Figure 4. Notice that the variables to be checkpointed must be registered with CPPC.

3) Final Code: CPPC uses the markup from Figure 4 to create a final version of the code that the C compiler can understand. Due to space limitations, that code is not shown here. However, the concepts involved in forming it are straightforward. For each checkpoint, line labels are inserted to mark the location of those checkpoints. Those labels are kept track of via an array that is populated when CPPC is initialized. Each label is assigned a unique id based on its location in that array, and when a call to checkpoint is made, the appropriate id is also stored in the checkpoint file.

When the program is restarted, the call to initialize takes care of repopulating the registered values that were in memory from the previous run. The code then jumps to the appropriate checkpoint label. This is achieved by using a "goto" command to jump to the line in the line label array referenced by the id stored in the checkpoint file. From here, the program proceeds as normal, continuing to checkpoint at the indicated locations in the program.

```
int main(int argc, char **argv)
{
   int fact;
   double curr;
   int i;
   #pragma cppc init
   fact = 20;
   curr = 1;
   #pragma cppc register
      ( fact, curr, i )
   for(i=1; i<=fact; i++)
   {
      #pragma cppc checkpoint
      curr = curr * i;
   }
   #pragma cppc unregister
      ( fact, curr, i )
   printf("%d factorial is %f",
          fact, curr);
   #pragma cppc shutdown
   return 0;
}
```

Fig. 4. The cppc markup of the factorial program

V. Evaluation

We have developed a test application to evaluate the performance of TALENT. The application contains 2,000 lines of C code and GUI developed using wxWidgets [17]. The graphical output of the application is sent through an SSH connection to a remote machine. Upon receiving a migration request, the application and its GUI are migrated from a Gentoo/Intel Xeon 32-bit machine to an Ubuntu 10.04.1/AMD Opteron 64-bit machine using environment migration and checkpointing.

The original migrations took a long time (about a minute), so we decided to time individual elements of migration. After breaking down the delays, we realized that synchronizing the filesystem took the longest time (98.7% of migration time). That should not be a surprise because when migrating, the entire filesystems available to the container must be copied to the destination. We decided to apply some optimization to the filesystem synchronization. In the optimized version, the filesystem is synchronized with the destination once before the migration. Then it is refreshed at periodic intervals by sending the differences to the destination. We chose 30 seconds as the synchronization interval. When a migration is requested, only the differences are sent to the destination. This simple optimization reduces the environment migration time to about a second. Figure 5 illustrates the performance of TALENT with and without optimization.

During the migration, the graphical output at the remote machine disappears for about 2 seconds. When the migration is completed the graphical output reappears on the remote terminal (now running on the second platform) without any user intervention because the state of the SSH connection is preserved.

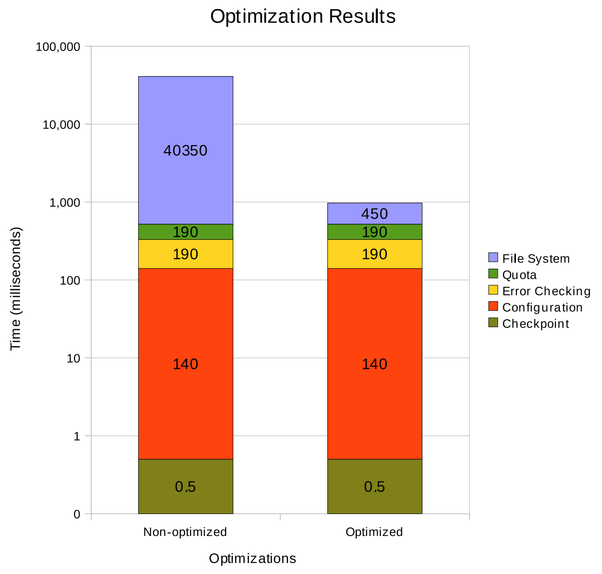

Fig. 5. TALENT's performance with and without optimization

VI. Related Work

Different process migration [8], [19] and virtual machine migration [5] techniques have been proposed in the literature. These methods are often used in high performance and cluster computing systems for load balancing and fault tolerance. However, they require homogeneous architecture and OS in order to preserve state.

Mead, et al. [12] discuss changes to software development life cycles for survivable systems. Sheldon, et al. [16] propose biologically inspired approaches for survivable cyber-secure infrastructures. Finally, Atighetchi and Pal [3] propose autonomic dynamic responses for survivable and self-regenerative Systems.

VII. Conclusion and Future Work

In this work, we described the design, implementation, and empirical evaluation of TALENT, a system that provides dynamic, heterogeneous platforms for mission critical applications. The current TALENT prototype is focused on providing high availability. More specifically, there is no guarantee that the migrated stated (persistent or ephemeral) is not already corrupted (integrity). As a future work, we plan to extend TALENT by adding sanitization and recovery capabilities. This would provide some level of integrity guarantees for an application under cyber attack. We also plan to augment TALENT with an attack detection engine which triggers the migration events. Finally, using attack graphs [9], we plan to integrate TALENT with an assessment framework so that the destination platform can be selected based on vulnerability and reachability analysis.

References

[1] Clustering in parallels virtuozzo-based systems. White paper, Parallels, 2009.

[2] Lxc man pages, 2010. http://lxc.sourceforge.net/index.php/about/man/.

[3] M. Atighetchi and P. Pal. From auto-adaptive to survivable and self-regenerative systems successes, challenges, and future. In *NCA '09: Proceedings of the 2009 Eighth IEEE International Symposium on Network Computing and Applications*, pages 98–101, Washington, DC, USA, 2009. IEEE Computer Society.

[4] G. Carl, G. Kesidis, R. R. Brooks, and S. Rai. Denial-of-service attack-detection techniques. *IEEE Internet Computing*, 10(1):82–89, 2006.

[5] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *NSDI'05*, pages 273–286. USENIX Association, 2005.

[6] T. H. Group. Hdf4 reference manual, February 2010. ftp://ftp.hdfgroup.org/HDF/Documentation/HDF4.2.5/HDF425_RefMan.pdf.

[7] I. Habib. Virtualization with kvm. *Linux J.*, 2008(166):8, 2008.

[8] R. Ho, C.-L. Wang, and F. Lau. Lightweight process migration and memory prefetching in openmosix. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1 –12, apr. 2008.

[9] K. Ingols, M. Chu, R. Lippmann, S. Webster, and S. Boyer. Modeling modern network attacks and countermeasures using attack graphs. In *ACSAC '09: Proceedings of the 2009 Annual Computer Security Applications Conference*, pages 117–126, 2009.

[10] K. Kolyshkin. Virtualization in linux. White paper, OpenVZ, September 2006.

[11] S. Lee, T. A. Johnson, and R. Eigenmann. Cetus - an extensible compiler infrastructure for source-to-source transformation. In *16th Intl. Workshop on Languages and Compilers for Parallel Computing*, pages 539–553, 2003.

[12] N. R. Mead, R. C. Linger, J. McHugh, and H. F. Lipson. Managing software development for survivable systems. *Ann. Softw. Eng.*, 11(1):45–78, 2001.

[13] T. A. X. Project. Xerces c++ xml parser, 2010. http://xerces.apache.org/xerces-c/.

[14] G. Rodríguez, M. J. Martín, P. González, J. Touri no, and R. Doallo. Cppc: a compiler-assisted tool for portable checkpointing of message-passing applications. *Concurr. Comput. : Pract. Exper.*, 22(6):749–766, 2010.

[15] E. Sarmiento. Securing freebsd using jail. *Sys Admin*, 10(5):31–37, 2001.

[16] F. T. Sheldon, S. G. Batsell, S. J. Prowell, and M. A. Langston. Position statement: Methodology to support dependable survivable cyber-secure infrastructures. In *HICSS '05: Proceedings of the Proceedings of the 38th Annual Hawaii International Conference on System Sciences*, 2005.

[17] J. Smart, K. Hock, and S. Csomor. *Cross-Platform GUI Programming with wxWidgets (Bruce Perens Open Source)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2005.

[18] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson. Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. In *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 275–287, 2007.

[19] G. Vallee, R. Lottiaux, D. Margery, C. Morin, and J.-Y. Berthou. Ghost process: a sound basis to implement process duplication, migration and checkpoint/restart in linux clusters. In *Parallel and Distributed Computing, 2005. ISPDC 2005. The 4th International Symposium on*, pages 97 –104, jul. 2005.