Distributed and fault-tolerant thread management for parallel computations

(draft version)

Alexei Iliasov (alexili@soros.kg)
Kyrgyz-Russian Slavic University
Toktogula 172 app 1, 720001
Bishkek, Kyrgyzstan
Tel: 996 312 502 363870
Tel/Fax: 996 312 434736

Abstract

Application of distributed computations becomes more attractive with the advances in computer networks. Lemick is an environment for automation of real-time distributed programming expressed in the terms of more traditional multithreaded programming. This paper gives a brief introduction into the implementation of Lemick distributed threads. A focus is made on the mechanisms employed to support fault tolerant execution environment.

Background

Distributed programming is often talked about in connection with such technologies as remote procedure call (RPC), remote agents (CORBA, SOAP, DCOM to name few), message queues (MS MQ) and message passing (MPI, PVM). They are similar in their aim to provide relatively simple and portable communication middleware. Lemick provides an environment that automates threads distribution. Lemick has a platform-independent bytecode interpreter and source-to-bytecode compiler to produce bytecode instructions from some high level language source. Lemick run-time environment (RTE) can to communicate with other RTEs through a network. They collectively map a task's threads onto the processors of the several independent systems interconnected by a network. Threads mapping is made absolutely transparent for a programmer. No changes to a source code are needed to make a multithread program run in a distributed mode. There are no special keywords, bytecode instructions or compilation modes to support threads distribution. The same program source and compiled bytecode version is used for a distributed, MMP, SMP or a single processor system.

A piece of code executing concurrently with the main program on a different host is called a *remote thread*. Lemick RTE creates remote threads by transferring parts of the main program's *bytecodes* by network to a remote system. Bytecodes are platform-neutral, compact and efficient representation of the program source. Program execution means either interpretation of the bytecodes by a *bytecode interpreter* or compilation of the bytecodes to the machine code with a *JIT compiler*. A remote system should provide standardized *communication interface, allocation environment* and optionally *language run-time environment*. Communication interface is needed to negotiate with other systems running a similar service. Allocation environment handles allocation request, implements *load balancing* and *failure detection* logic. Language run-time environment

implements a bytecode interpreter or a JIT compiler. A remote system without language run-time module is called *gateway system.* Gateway system does not allocate threads on itself but delegates thread allocation requests to other systems. Though such systems can not run programs locally they still have full control over the code running remotely including failures detection and recovery. One important requirement to a gateway environment is that it should run in a single thread. Gateway environments are very convenient to use on the hosts that serve as network gateways. Such hosts sit in several separate networks that are not routed to each other. Networks gateways are usually already busy enough providing low-level network services thus gateway environment should be fast and small.
Another important application area for gateways is enabling restricted computing environments such as hand-held computers and embedded systems to use distributed programming. Since gateways are small and simple they are relatively easy to port or rewrite. Also absence of support for multi-threading in many operating systems used in restricted environments prohibits porting of the language run-time environment.

Lemick threads distribution system was developed to be very scalable and to support large number of hosts participating in a single task. This is achieved through the structuring of the hosts interconnections. *Home system* is the system where the task was started. *Remote system* is a system used to allocate one or more threads of the task from the home system. Thread allocation means that a piece of code received from the home system will be executed on the remote system under the full control of the home system. The remote system agrees to not interact with the allocated thread except as commanded by the home system. *Zone of authority* of a system is a set of the hosts to which the system may *delegate* thread allocation requests. Delegating allocation request means passing the allocation request and all the corresponding data to a system from the zone of authority. For the system that accepts a request, system delegating the request appears as a home system. Chain of the hosts between real home system and real remote systems are called *mediators*. A mediator is responsible for receiving and retransmitting packets between a home system and a remote system to which a thread allocation request was delegated. A home system can (but does not have to) establish connection to its neighbors only. List of the neighbors is read at the startup from the configuration file. Systems may accept thread allocation request only from their neighbors. However actual allocation may occur on a system that is not a neighbor of the home system. This is a consequence of the requests delegation feature. Allocation requests delegation is also an instrument of the load balancing algorithm that strives to make optimal use of the available resources.

Failure detection mechanisms

Lemick RTE employs special logic to discover dead remote hosts or neighbors. Similar approach is used in several modern routing algorithms [1] and is known to be cheap and efficient in terms of network load. Every host emits small packets called "*Hello*s" to the neighbors to let them know that system is in operational state and there is a network route to it. This method is known as *heart beating*. Hello packets always use cheap unreliable delivery. Lemick RTE uses two implementations of the heart beating - with uniform and non-uniform intervals. In the first case systems emit Hello packets in the uniform intervals without any payload information. In the later case packet also includes *hold-out* *time* value. Hold-out time specifies in what time to expect the next Hello packet from the system. No answer or acknowledgement of the delivery is expected for Hello packets. If a remote system is silent for several intervals it is considered dead. Though protocols that use reliable delivery can detect and even tolerate many network errors they do really poor job when it comes real-time computations. For example typical time out value for internet routing protocols is about two minutes. A remote system could be dead already for a long time when underlying software reports about the problem.

To pass information between the systems Lemick RTE uses *message* packets and *stream* packets. Stream packets are used to effectively handle streaming input/output redirected from a remote to a home system. Stream packets are processed immediately upon the arrival and are never queued. For example "Hello, world!" program executed in a remote thread will use one or more stream packets to pass the string literal to the home system. Message packets are queued and may be processed asynchronously. There are two major types of message packets. *Notifications* are packets with non-empty data section sent from one system to another. Notifications do not require any answer from the receiver. Notifications always use reliable delivery, which means that their delivery is guaranteed and they are delivered and processed in the proper order. *Request* packets are much like notification packets. The difference is that the RTE will block a sender of a request packet until a *request answer* packet is received. Usually *request answer*s will bring some information from the answering system.

Failure recovery mechanisms

For non-critical classes of exceptions Lemick RTE defines *default exception handlers.* They are invoked if no user-defined handler could be found for a raised exception. Default handlers usually execute some clean up code and try to resume execution in the context of the raised exception. If there is no default handler for an unhandled exception a program is halted.

Here comes a brief description of some interesting failures that are detected and somehow handled by Lemick RTE.

Unrecoverable failure on a home system. Active multi-home redundancy can be used to tolerate certain hardware faults [3], [4]. *Multi-home* redundancy means that a task at some moment has more then one home system. When a task is started on several homes it is an *active* multi-home redundancy. Such tasks have one main and several additional home systems. All the homes are initially indexed starting from zero for the main home. Copies of the packets received from a remote system are resent to the additional homes. Lemick RTE translates packets headers so that they appear as a packet from the corresponding remote systems of the particular additional home. A host can not serve as both an additional home and a remote system. This prevents from simulating redundancy within a single host. If the main home is unreachable a home with the lowest index becomes the new main home and indexes of the homes are decremented by one. If there is neighbor of the new main home that is not used for threads allocation it becomes a new additional home. Since failure detection is based solely on Hello packets remote host failure can not be discovered immediately. This is a major rationale for using reliable delivery for all informational packets.

Home system administratively stops the service. A reason for that could be a scheduled power off or transient hardware or software failures. In both cases the RTE will remain stable for some time. This time should be enough to move task home to some other system. This kind of multi-home redundancy is called *passive*. Currently Lemick RTE can only kill and restart the program on a new home system. Better realization would require implementation of logging or checkpoints [2] that is rather expensive for a distributed environment. Unlike active redundancy, passive redundancy does not depend much upon the quality of the failure detection mechanism.

Unrecoverable failure on a remote system. A loss of a network connection to a remote system is a typical event of this kind. Lemick RTE may silently restart *idempotent*[1] threads allocated on a failed remote system. Non-idempotent threads are considered *lost* and an exception is raised in their parent thread. If a lost thread had children they are also killed in some time. For still living children of the lost threads the failed system plays a role of a home system. That is a case of the previous paragraph. Hello packets are used to detect this kind of failures.

Remote system detects failure of a home system. If a remote system discovers that a home system which requested allocation of one or more threads is dead it does nothing but kills all the threads allocated by the request from the dead system. All the delegated allocations requested by the dead system are also killed. Note that event triggering the actions described above is the same as in the first paragraph. The only difference is in the role of the system within the particular task. Also note that a remote and an additional home system can be located on the same host within a single task, so there is no ambiguity in recovery action.

Unhandled exception in a thread on a remote system. A thread in which the exception was raised is suspended. A remote system sends request to a home systems to handle the exception. The request includes serialization of *Exception* object for the raised exception. The home system propagates the exception to an outer scope (parent thread, for example) and sends a request answer when the exception is handled.

Exception is raised during SEND-RECV rendezvous. SEND-RECV statements may work in both blocking (synchronous) and non-blocking (asynchronous) modes. If an error is detected in blocking mode it is propagated in the contexts of SEND and RECV. In non-blocking mode SEND and RECV are not paired so they detect errors and recover asynchronously.

References

[1] Idempotent task is a task that has no side effects and which return value (if any) is determined only by the values of the formal arguments. For example function *f(x)* that calculates *x*x* is idempotent, while function that returns time-of-the-day not.

[1] Routing TCP/IP, Volume 1. Jeff Doyle. Cisco Press. 1998.

[2] Hypervisor-based Fault-tolerance. Thomas C. Bressoud and Fred B. Schneider. 1995.

[3] Framingham Corporate Center. Fault tolerant CORBA Using Entity Redundancy. 1998

[4] Fault-Tolerant Sequencer: Specification and an Implementation. Roberto Baldoni, Carlo Marchetti and Sara Tucci Piergiovanni. 2001

[5] RFC N3080