Chapter 9
Programming for Distributed Computing: From Physical to Logical Networks

Christian Scheideler and Kalman Graffi

The first programming languages predate computers and were mostly used as a form of mathematical reasoning. With the advent of modern electronic computers, programming languages became prominent as tools to specify and control the behaviour of these machines. A programming language is an artificial language that makes use of the functions that can be performed by a machine and allows one to express algorithms precisely. Early languages (see Chap. 4) were Fortran (1957) and Algol (1958), which were used for numerical computations. Cobol was issued in 1962 and optimized for business applications. Many other influential languages emerged from the late 1960s to the late 1970s, among them Simula (which introduced object-oriented programming), C, Smalltalk, Prolog (the first logic programming language) and ML (realizing a polymorphic type system on top of Lisp). Each of these languages spawned an entire family of descendants, and most modern languages count at least one of them in their ancestry.

The capabilities and characteristics of a programming language define the type of computer processes that are contemplated. The von Neumann computer architecture (Chap. 5) established a basic model for sequential computation processes. It postulates (Chap. 4) a computer organization based on a single separate central processing unit (CPU) and a separate memory unit accessible to the CPU. Having single or multiple CPUs or memory units introduces a classification of various types of computers and related programming approaches. Due to trends in the design of computer chips, nowadays, multiple CPUs and multi-threading (sequences of computation steps) on one chip are de-facto-standards in computers. This trend emerged in 1963 when the first time-sharing systems appeared.

Multi-processor machines introduced various challenges for the concurrent access of programs to the shared CPUs and memory. With the advent of the Internet era and the trend towards cluster computing (Chap. 8), the distribution of computing devices

C. Scheideler (✉) • K. Graffi
Department of Computer Science, University of Paderborn, Paderborn, Germany
e-mail: scheideler@upb.de; graffi@mail.upb.de

E.K. Blum and A.V. Aho (eds.), *Computer Science: The Hardware, Software and Heart of It*, DOI 10.1007/978-1-4614-1168-0_9,
© Springer Science+Business Media, LLC 2011

and the memory widened even further. A cluster of computers offers many distributed computational and memory resources, more than an individual computer. Operating systems were organized to provide services during the execution of a program, such as links to various built-in system subroutines. This is called the *runtime env*ironment. With a suitable computer language and runtime environment, distributed resources can be harnessed and conveniently used on single applications. For more details, see Chap. 8 on high performance computing.

In this chapter, we elaborate on general principles of distributed and parallel programming and discuss specific solutions for distributed computing in physical networks as well as in application-layer networks, which are also known as *overlay* networks.

Distributed and Parallel Programming

A *distributed* or *parallel* computer system consists of a family or set of autonomous computers organized to interact so as to cooperatively execute a computation, such as a cluster (Chap. 8). The autonomous computers offer either single or multiple computational resources. In the case of each autonomous computer having also a private memory, we speak of a *distributed* system. In this case, the computers interact through a network in which they exchange messages and synchronize their actions. (See the HPCC Chap. 8). If the autonomous computers have access to a *shared memory,* we use the term *parallel* computing. In this case, communication is performed via the shared memory.

The motivation for the use of parallel or distributed systems is two-fold. First, an application may impose distributed characteristics by creating and consuming data in differing physical locations. Such a characteristic is, for example, found in the various email applications or in typical (client-server) consumer-provider based applications. The second reason for distributed/parallel computation is motivated by the practical benefits in comparison to the use of a single computer. In comparison to a single computer, a cluster of computers may be extended in number and thus extended in the resources available. There is no intrinsic scalability limit in the clustering of the computers. *Server farms* of up to 100,000 computers exist. Through scalable and redundant (software and hardware) architectures, the damage effects of a single point of failure can be eliminated. However, in order to realize the full potential of a distributed/parallel system, a programming language is needed that has primitives for specifying the effective use of the available resources.

Levels of Parallelism

As discussed in Chaps. 3 and 4, a *program* is a list of instructions to be executed on a computer aiming at performing a specific task. In its simplest execution mode, its

instructions are run sequentially on a CPU, one after another, storing intermediate and final results in a single memory. However, to use the power of a large number of computers, the program must be split up into smaller parts that are run *concurrently,* or as we say, in parallel. Various levels of parallelism exist that define the *granularity* by which a program may be split up. The highest level of granularity is on the *program* level. On this level, various programs may be run independently on multiple computers, but no tasks within a program may be shifted or run on other computers. The next level of granularity supports parallelism in the execution of *procedures*. In this case, instruction sets, encapsulated in procedures, may be transferred (or *called)* to ex*ecute* in parallel at remote computers. At this level, coarse parts of programs may be placed so as to execute on more suitable computers. At the *instruction* level, individual operations (e.g. arithmetic operations) may be run in parallel to more efficiently use the available resources. At a lowest level, the *bit* level, the internal execution of a single instruction is performed in parallel, e.g. on a multi-core CPU or a GPU with many *stream* processing units. With this grain of parallelism, individual atomic operations can be speeded up.

While very fine-grained levels of parallelism in computer programs are best performed in hardware, the parallelism of instructions and procedures is best exploited in software by distributing the operational load of the execution of a program over various *cores* or computers. To obtain full access to the distributed resources, the procedures or instructions may be either dispatched or called manually by the programmer or may be run automatically by the runtime system environment. Both of these approaches to parallelize applications have been heavily investigated in the literature. The automated approach requires recompiling programs with special parallel compilers in order to enable their parallel execution. This is convenient for the programmers as existing programs do not need to be adapted and no further work is needed. However, automated parallelization often lacks efficiency, as it is very challenging for a compiler to optimally split the program into parallelizable tasks. Using the second approach, the programmer needs to learn how to deal with parallel programming patterns and explicitly apply the parallelization instructions to exploit parallelism. While this approach is more time-consuming in the creation of the programs, it usually results in a much more efficient parallel execution of the code.

Tasks in Parallel Programming

For both approaches, the implicit and explicit parallelization paradigms, a set of management tasks needs to be executed by the runtime environment in order to enable the parallel execution of a program. According to Kasim et al. (2008) the set of available computers (*workers*) needs to be managed, the *workload* needs to be partitioned into tasks, the *tasks* mapped to workers (computers) and the intermediary results need to be synchronized. The first step comprises the management of the workers. It is needed to

supervise and monitor the resource utilization of the workers as well as the connectivity and contact details of the workers in the case where they are distributed. On the other hand, the program must be split into sections of concurrently runnable *processes* and prepared for parallel execution. This is a non-trivial programming design problem as the semantics of the program may change subtly if certain processes are executed prior to others. This step is either done manually by the programmer or automatically by the runtime environment. Given the workers and the individual program *processes*, a *mapping* is needed. The mapping from processes to workers must take the abilities of the workers into account as well as the requirements of the processes. In the case of shared memory, the tasks in the memory are accessible from all CPUs. In the case of a distributed system, the processes are either pre-deployed and remotely accessed or need to be distributed in advance to the corresponding workers. Finally, the results of the calculations of the processes need to be synchronized and joined. For distributed computing all the processes need to be suitably addressed. In the following, we present how these tasks are addressed in physical networks. Specifically, we present concepts for parallelism on computers and networks, give examples and introduce selected programming languages that ease the distributed programming. (See Chap. 8 for related discussions.)

Physical Computers and Networks

In this chapter, we distinguish between distributed systems based on actual physical networks and virtual systems based on logical networks. In the first case, computers or *cores* are locally close to each other and connected directly. This case also comprises the hardware-based multi-core computers, in which the CPUs are also closely linked. In this case, the CPUs and memory are close to each other, so the main questions are related to how the access to the resources is managed. We discuss the main questions related to distributed systems based on logical networks in the next section.

In order to utilize the resources several programming approaches are in use in practice. One main classification criterion is the involvement of the programmer. On the one hand, the programmer may explicitly decide which instructions should be run in parallel. On the other hand, this decision may be handled implicitly under the assumption that the entire code is parallelizable. Given that the assumption is true, the runtime environment may optimize the code execution on its own using various programming concepts outlined below.

Concepts of Parallel Programming

Next, we describe a selected set of concepts for *parallel programming*. In the early days of parallel programming, single CPU devices supported subroutines called *co-routines*. In order to use these, the programmer had to actively trigger a co-routine

operation. Co-routines are subroutines that can be run in parallel and may be paused. However, this approach is limited to single-core devices and cannot be extended to multi-core computers.

The *fork and join* approach is another of the earliest parallel programming concepts. The names are suggested by flowcharts. The *fork* operation in a program marks the beginning of a parallel process, while the *join* operation synchronizes the results. Applying the fork operation in a UNIX OS environment copies the process that is forked and enables each process to identify its status through checking its process identifier. Forked processes are run in parallel and combine their results in a matching blocking *join* call.

The *remote procedure call* (RPC) is an approach to integrate remotely located procedures. Procedure calls for remote and local code have the same syntax. RPCs are resolved transparently for the programmer. The runtime environment of the programming language offers client and server *stubs* as local code. These stubs are in charge of managing the network connections, marshalling and de-marshalling the messages and managing blocking operations.

While the previous concepts require the programmer to use parallelism explicitly, *implicit parallelism* denotes a different parallelism concept. This concept assumes that every *process* can be run in parallel except for parts called *critical codes*. Thus, the execution of a process may be interrupted and run on a different core or computer. In order to protect *critical code* parts from being disrupted, the programmer may set critical code in a *synchronized blo*ck of code.

Examples of API's for Parallel Programs

Most of these concepts have been implemented, deployed or discussed in the literature. In the following, we introduce successful APIs that gained a lot of attention in recent decades.

The software tool *parallel virtual machine* (PVM) is an API designed to allow a (heterogeneous) network of computers to be used as a single distributed parallel computer platform (as elaborated in Sunderam (1990)). Using PVM, program *tasks* or *independent threads* of instructions may be run in parallel on the virtual machine using explicit remote invocation. Through explicit message passing based on task identifiers, the tasks may communicate with each other. The PVM author group offers libraries of such parallelizing devices for the C, C++ and Fortran languages and wide support for a large set of devices.

The *Message Passing Interface* (MPI) is explained in Pacheco (1996) and is a de-facto standard for developing high performance computing applications on parallel and distributed memory architectures. (See Chap. 8.) It specifies message passing (MP) operations for *point-to-point* message passing between pairs of processors and also for *collective* message-passing between groups of processors. It provides libraries of MP subroutines for C, C++ and Fortran as well. MPI requires the programmer to mark individual processes (tasks) in the program and map the

processes to workers (computers). The user of the program may define how many processes are assigned for the execution of a program. For common modes of communication between processes, MPI supports point-to-point and collective communication based on message passing. In addition, a barrier analogy is used to synchronize the processes. The barrier operation requires all specified processes to pass the barrier and thus to finish their tasks before the execution of the processes can continue. Thus, programming entails some intricate planning of the paths of concurrent execution.

Several theoretical models for distributed programming concepts have been presented in the literature. Lipton and Sandberg introduced an abstract machine termed *Parallel Random Access Machine* (PRAM) in Lipton and Sandberg (1988) which models the various types of concurrent read and write operations in a shared memory system. It abstracts from synchronization and communication issues, but gives a tool at hand to investigate the effects of various distributed programs.

Bulk Synchronous Programming (BSP) was introduced by Valiant as a model taking the synchronization and communication overhead into account. BSP introduces three stages to model parallel computing. In the first stage, a set of processors with local memory is assumed. These processors perform local operations and do not interact. In the second stage, communication between the processors takes place and results are exchanged. Finally, in the third stage the processors are barrier synchronized by waiting until all processors finish a particular communication. As the two last stages are expensive, the idea is to perform as much local computation as possible in the first stage so that the overhead generated by the other two stages is insignificant. Obviously, too much overhead would incur delays that could cancel any speed-up expected from concurrent execution. This is true of any parallelization technique. The stages are continuously repeated until the program terminates. This model has been intensively discussed and extended in the literature, and several implementations exist, (see Bonorden et al. 2003). A few examples are the logP, QSM and HMM models.

Another formalization of interactions in a concurrent system is given by Hoare, and termed *Communicating Sequential Processes* (CSP), (see Hoare 1978). Such a formalization helped to specify and verify the correctness of various concurrent programs.

MapReduce was introduced by Google in Dean and Ghemawat (2008) to facilitate the processing of large data sets on a set of distributed computers. Libraries of the software framework have been developed for various programming languages. In the *map* step, a problem is split into smaller sub-problems and assigned to a set of workers, which themselves may split the assigned problems further, creating a sub-problem tree structure of the problem. Each worker reports its results to its father, which combines in the *reduce* step the individual results of its children workers to its own result. With this approach, data-intensive and loosely coupled computational problems can be solved very efficiently by a large set of workers. This scenario is often the case in cloud environments like those in Google's or Amazon's server farms.

While these examples represent certain standards or models, another approach to parallel programming is defined by the set of available concurrent (and parallel)

programming languages. With a specific parallel/concurrent programming language at hand, a programmer can exploit parallelism more efficiently. Several such programming languages have been introduced in the last few decades. We present some of the most influential ones below.

Concurrent Programming Languages

In the early days of software engineering, the discipline of programming was mainly concerned about the increasing complexity of the programs. Early structured programming languages like *Pascal*, *Fortran* and *Cobol* introduced *structured programming* as a method to create reusable, dependable code. However, they were limited to numerical problem solving in their practical applicability. Starting with 1962, the idea of information hiding and abstract data types shifted the focus from numerical programs to object-oriented programming, which was first used in the *Simula* language.

Concurrent programming languages were first discussed with the new time-sharing machines and later since 1975 with the advent of the personal computers and their organization into networks. Some of the object-oriented programming languages were extended to support the new trend of parallel programming, such as *Smalltalk* (1972) or *Ada* (1973). Previous programming languages did not sufficiently support the parallel execution of programs. *Occam* was introduced in 1983, following the strict formalization of Hoare's CSP. *Erlang* was introduced by Ericsson in 1986, supporting concurrent programming using an *actor model*, which relies on message passing between processes rather than shared variables. However, with the Internet era and the World Wide Web, *Java* started its success story in 1996, when it was introduced and supported as the programming language for the WWW. (See Chap. 4.) Due to its application field and usability, Java had a large impact both in industry and academia. Several extensions for secure distributed programming emerged, leading to new programming languages such as *E*. The language *E* originates from Java and uses message-passing, event loops and promises for managing the concurrency in the programs. Another programming language that relates to Java is *Clojure*, which runs on the Java Virtual Machine. However, *Clojure* is a functional programming language, combining the benefits of *LISP* and *Java*. One of the youngest programming languages is *Go* developed and announced in 2009 by Google. *Go* is similar to *C* but with structures for concurrent programming. *Go* is inspired by Hoare's CSP. However, unlike *Occam*, *Go* does not support verifiable or safe concurrency.

In the history of concurrent programming languages, specific patterns recur. Message passing is a de-facto standard, as the alternative shared memory, mutexes or semaphores are technically infeasible or inefficient. As an example, we introduce the programming language *Erlang* to show the main principles for concurrent computing based on message passing. (See also Chap. 8.)

A Brief Introduction to Erlang

Erlang was developed around 1986 by the Ericsson company as a programming language that supports concurrent programming, focusing on highly available programs in the area of telecommunications. Its main approach to increased availability is to strictly isolate the individual processes in a program, avoiding any means of shared memory. The processes are, however, allowed to communicate using message passing, both with processes on local cores but also with processes on remote machines. An essential requirement for safe message passing is the usage of *pure protocols*, which do not use any pointers or information which might be bound to a specific machine. Fault tolerance is further reached by allowing machines and processes to crash. Instead of trying to catch any possible exception in a process and to react to it internally, an invalid process is left to crash. However, the crash of any process is communicated and detected by a monitoring process, which initiates actions upon the observation of such a failure. For that, the monitoring machine needs sufficient information to decide on the appropriate next steps. In particular, it needs information on the cause of the crash. Thus, fault tolerance is reached with replication and a safe-to-crash approach. This approach was considered inefficient back in the 1980s, especially due to the limited bandwidth and duplication of program state information (instead of using a shared memory). Nowadays, the main bottleneck in the execution of a program lies in missed cache hits in the case that illegal pointers are used. The approach of duplicating the program information for message passing fits the current trends of computer system architectures.

An *Erlang* program typically consists of thousands of processes, divided into *worker* and *monitoring* processes. Worker processes have links to monitoring processes, which form a error-propagation channel in the case that the worker process fails. This classification of process types is described in the *Erlang* Open Telecom Platform system, which suggests monitoring trees that hold a protocol that is initiated once a failure is detected. A failure of a working process is interpreted as the inability to perform a desired computation. Thus, a new (easier) approach for solving the problem is typically instantiated.

Erlang offers in its first implementation a *Prolog*-like programming syntax with *variables, atoms, tuples* and *lists*. In order to support concurrent programming, message passing is modelled by a "P ! M" command, meaning that the message *M* is sent to the process with the process id *P*. Knowing the process ids is a must in *Erlang*. The receive operation *receive* is called in a non-blocking way at the addressed process. It matches the received message with a list of patterns and initiates the corresponding action upon a match. In order to fork a process to run in parallel to the calling process, the *spawn* command is to be used. It initiates a new process and reports its process id, which can then be used as a communication address.

Error detection of remote or local processes is set up using a method called *link (PID)*. This call links the current process to monitoring the behaviour of the process with process id PID. In order to receive these out of band messages, the monitoring

process is flagged as a system process using the command *process_flag(trap_exit, true)*. Upon an irregular termination of the monitored process, an error message is created containing the keyword *Exit*, the process id of the broken process and a reason, ('Exit', PID, Reason). This message is passed to the monitoring processes and evaluated there. This simple approach allows to create large and complex applications. Message passing is used both between large components, but also within a large component between the smaller components it consists of. Any component of an application can be tested whether it produces expected results. Components producing unexpected or false results can be opened and tested recursively until the root of the failure is found.

Erlang has been in use at Ericsson to operate ATM telecommunication switches in a reliable and fault-tolerant manner. With the Open Telecom Platform, a set of libraries has been created that provides all essentials of an application middleware package, easing the use of Erlang for large and complex programs. Parallel to the advent of Erlang, further trends of parallel and distributed computation in the Internet have been observed. These trends were not influenced by programming languages, but more by novel distributed architectures and principles. *Logical networks* were formed, granting direct access to distributed resources, allowing new approaches to parallel and distributed programming.

Logical Networks

For a long time, hardware (physical) networks, realized either on a chip or among a set of hardware devices, were used for distributed programs. The main assumption was that all of these devices were controlled by the same person or institute. (See Chap. 7 on clusters.) Later, in the Internet (Chap. 7), a client-server (software) architecture was put in use to support such popular applications of WWW as FTP and email. This is a logical network overlaying the physical one. The main idea was that many users access a centrally hosted server and exchange their information over this central mediator. Local code and functional roles were more and more shifted to servers in the Internet, leading to weak and simple clients on the user side. This logic network approach allows the operation of specialized machines for certain tasks to buy and sell computation services in the Internet and to increase both efficiency and productivity.

The concept of *service oriented architectures* (SOA) was introduced in the late 1990s as an approach to organise software architectures i.e., logic networks. Its main idea is to map business processes to *workflows*, which are decomposable into smaller service units. Individual services may then be either programmed by the user himself or bought in a (not yet existing) global marketplace for services. Through the clear separation and specification of single services, a marketplace may be born, leading to professional and well tested service components as well as flexibility in the design of system architectures. A further extension of this idea came about recently with the advent of c*loud computing*. (See Chap. 8, Appendix.)

While SOA is limited to the hosting and execution of remote code, cloud computing offers a wider set of services. As already remarked in Chap. 8, Appendix, cloud computing can offer software as a service (SaaS), platform as a service (PaaS) and infrastructure as a service (IaaS). We elaborate below.

At the end of this chapter, we summarize the research and industrial efforts in the area of *peer-to-peer* (p2p) networking. p2p systems are distributed architectures created through establishing a logical network between peers typically connected to each other via the Internet. The peers are mostly autonomous computer installations under control of individual users. Thus, it is challenging to provide the functionality of a desired application with a desired quality. This is a developing field of distributed programming.

Service-Oriented Architectures: A Survey

Service oriented architectures (SOA) extend the idea of open interfaces and RPCs in such a way that the components or services to be used are offered in a world-wide marketplace. Thus, productive and well-programmed services may be rented and used to compose complex software architectures. SOA promises to make the creation of software architectures more flexible and thus allow a fast adaptation of these architectures to business needs. Business workflows are a main driver in SOAs. These workflows define the steps in business procedures controlled by computers and are partitioned into individual services, which are composed from the set of available services in the service market place. Individual services may be exchanged for cheaper or more productive service implementations. In order to dynamically do this, a set of automated actions need to be performed. A service provider needs to *publish or register* his service implementation at a service directory like UDDI (Unified Description, Discovery and Integration). The service consumer (client) needs to *find* a service (in the directory) that matches its needs. Match making is a challenging task even though service descriptions use standardized web service description languages (WSDL). These are not always explicit on defining the semantics of a service. Once a suitable service is found, it is *bound* to the service consumer. This is done by the service directory delivering a contact address for that service to the service consumer. As a final step, the service is *executed* in a typical RPC manner by sending the input values in a well-formatted manner, typically using the protocols SOAP (Simple Object Access Protocol) or REST (Representational State Transfer).

While SOA does not introduce a new programming language for parallel or distributed computing, it parallelizes the programming and service consumption itself. Programs or program components do not have to be written by the same person or loaded manually. Services are searched and bound automatically to the runtime file of the program, picked and paid at an internal or global market place. The service composition can be optimized according to various characteristics, ranging from the costs to the service level agreement details. The quality of service

provided by every service is specified, as well as the API and costs for using it. While a SOA may be easy to implement on the scale of an organization, a global market place is still to come. Many users and enterprises hesitate to release their services to avoid showing internals of their business workflows. Also, enterprises fear to use the services of other providers in order not to reveal internal workflow information by transferring revealing input values in a service call. An extension of the SOA idea from software services to platform and infrastructure services has recently had a large impact under the term "the Cloud". See Chap. 8 Appendix.

Cloud Computing-A Survey and Critique

Cloud computing (Chap. 8 Appendix) emerged consequent to the trends of SOA, as a method of on-demand and pay-per-use services with regard to a wide set of resources. While SOA focuses only on software as a service, with the advent of *grid computing* (large networks of existing distributed computer resources) and *utility computing* platforms for application hosting or infrastructure resources themselves can be reserved and used nowadays over the Internet. A great driver for these trends lies in the *virtualization* of hardware devices, allowing systems to host or port several instances of software programs. Thus, with the demand and access patterns for a specific software or hardware service, the corresponding hosting resources can now be increased or decreased on the fly, without the interruption of the running program.

The emergence of Web 2.0, an extension of the classical World Wide Web that allows interaction with websites (Chap. 7) has accelerated the shift of personal user data to the Internet, where it could be modified or hosted using various novel Web 2.0 applications. *Software as a service* (SaaS) as one aspect of the cloud provides applications in the Internet, both to be used by users directly (e.g. Google Docs) or over specified APIs (e.g. Open Social). Having the user data in the Internet (e.g. Facebook) allows the direct manipulation and the central hosting of applications for it (so-called *social networking*).

A further aspect of the cloud comprises the offering of platforms, or *platform as a service* (PaaS). In this case, vendors offer a rich set of functionality to support the life cycle of individual services to be hosted in the Internet. The life cycle support ranges from application design and development, testing, deployment and hosting, as well as background services like storage, monitoring and accounting. Platforms offer Web service designers the freedom to offer their services and pay only for the consumed resources, for which they also can charge their customers. This process liberates the designers from predicting and reserving hardware resources that their application might use in peak times. The elasticity of the cloud, i.e. adapting the resource provision to the demanded level, is a key benefit for both the application designers, operators and users. However, several limitations on program structures were defined by the vendors. Computation State is considered harmful, as it cannot be elastically ported on the virtualized computers.

From a commercial point of view, most cloud providers offer very competitive prices in comparison to self-operated server farms. This comes at the danger of lock-in and being bound to a specific cloud vendor. It still remains challenging and work demanding to change the programs hosted on one cloud to be runnable on another cloud. Due to differing commercial interests, cloud vendors aim at hindering the support of interoperability.

A third layer in cloud computing offers more freedom to its users. *Infrastructure as a service* (Iaas) offers core computational and storage services on demand, leaving the usage and management of these resources up to the service consumer. Examples of the various layers of cloud computing are the Google App engine, Amazon's Elastic Compute Cloud (EC2) and Microsoft's Windows Azure Platform.

p2p Networks-a Survey and Critique

While a small set of cloud players were trying to build centralized server farms to offer any kind of services a user needs, an orthogonal trend emerged with the advent of Napster, BitTorrent and Skype. This trend of peer-to-peer (p2p) networks provides services and applications through the creation of logical networks and distributed mechanisms mainly or solely using the resources of the consumer devices. In contrast to the previously mentioned approaches, p2p systems consist mainly of user devices that are autonomous and out of reach of a provider. Thus, the resources are freely available, as are the users, but are unreliable and of fluctuating quality. In addition, due to the large number of autonomous nodes, networks operating on p2p resources must be *self-organising* and consider the fluctuation of node resources and presence. This paradigm has had a large impact in the area of file sharing applications. In 1998 *Napster* offered a centralized index which acted as a publish/subscribe platform to share and search for information files among the users. The subsequent file transfer after a positive match was performed directly from peer to peer. Due to the vulnerability of the centralized index, subsequent file sharing applications used a logical network, an overlay, to interconnect the peers and to implement the search for desired files or objects.

A set of overlays has been proposed both in academia and open source communities. This set can be classified into structured and unstructured overlays. Structured overlays, like *Chord*, *CAN* or *Pastry*, place the objects stored in the overlay by a method based on their object IDs according to a predefined structure. Thus, the insertion of objects requires some time, but the retrievability of these objects is guaranteed and the lookup time is low. In an unstructured overlay, like *Gnutella 0.4* or *GIA*, objects remain at the peers, thus produce no additional overhead for insertion, but the search for these objects is rather costly. In the worst case, all peers in the network have to be contacted to find a desired object.

A further classification aspect of overlays is whether all the peers have the same roles or whether there are some peers with special roles and more responsibilities. Super-nodes are used typically in hybrid overlays like *Gnutella 0.6* to maintain the

state of normal nodes. Most overlays are flat, meaning that all peers are organized in only one overlay. Hierarchical overlays which consist of various protocols for various layers of the network, e.g. like in *Omicron*, are rare. Most popular overlays nowadays implement a *distributed hash table* (DHT), which was first introduced in the Chord paper by Stoica et al. (2001). The main idea is to split up a hash table and distribute it over all peers. Maintaining the links to the neighbouring parts of the hash table as well as shortcuts in the hash table to peers in exponentially increasing distances allows traversing the hash table consistently and quickly to find the desired entry. Dabek et al. (2003) a common API for structured p2p overlays which extends the common lookup functionality of hash tables. Several overlays follow that principle, like *Chord*, *CAN*, *Pastry* or *Tapestry*. With the distributed nature of the overlay, the most critical challenge to be addressed is the quality fluctuation of the peers, which requires mechanisms to repair the structure of the overlay all the time.

p2p networks offer access to the wide set of resources that comes with the user devices. In contrast to centrally hosted applications, even the cloud, which reaches its limits with the increase of users, p2p systems benefit from an increase of the number of users, as the pool of resources increases as well. p2p networks are mainly used for file sharing applications, in Skype for free Internet telephony or in some applications, like *PPLive*, for live and on-demand video streaming. However, a broad service offer like in the cloud is not yet feasible in p2p networks due to the limited control of the quality of these networks. The quality fluctuates with the quality of the resources offered by the users. This is a characteristic that is unsuitable for most productive applications. Although there is a vast amount of resources to use and to harness, research has not yet devised the tools and mechanisms to contend with the quality fluctuation and to provide reliable quality based on the user resources contributed to the p2p networks.

Conclusions

Distributed programming emerged with the advent of multicore and networked computers and reaches out to the trend of globally distributed software development. Nowadays, with more and more computational resources distributed on a single machine, at home locations, in widespread enterprises or even over the world, tools and paradigms are needed to handle the desired programs and problems in a way that harnesses the potential of these distributed resources. Distributed programming languages give a tool at hand to create suitable programs that are parallelizable, distributed runnable and still fault-tolerant and reliable. In this chapter we described the characteristics and classifications of distributed programming and briefly introduced the features of the programming language Erlang. In the future, however, system architecture will play a larger role than it did in previous decades. With globalization, software projects are globalizing. Software components are envisioned to be bought and bound on a global market place, corresponding operational platform

resources are rented on demand to run those services and distributed programming emerges to become the art of distributed service composition. On a small scale, however, the craftsmanship and art of distributed programming needs to be further investigated and improved in order to create the building blocks of society's key tool for business prosperity: the IT industry.

Reference

O. Bonorden, B. Juurlink, I. von Otte, and I. Rieping. The Paderborn University BSP (PUB) library. *Parallel Computing*, 29:187–207, February 2003

F. Dabek, B. Y. Zhao, P. Druschel, J. Kubiatowicz, I. Stoica: Towards a Common API for Structured Peer-to-Peer Overlays. IPTPS 2003:33–44

J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM*, 51(1):107–113, 2008

C. A. R. Hoare: Communicating Sequential Processes. Commun. ACM (CACM) 21(8):666–677 (1978)

H. Kasim, V. March, R. Zhang, and S. See. Survey on Parallel Programming Model. In, *Network and Parallel Computing*, volume 5245 of *Lecture Notes in Computer Science*, pages 266–275, 2008

R. J. Lipton and J. S. Sandberg. PRAM: A Scalable Shared Memory. Technical Report, TR-180-88, Princeton University, Dept. of Computer Science, Sept. 1988

P. S. Pacheco. *Parallel Programming with MPI*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996

I. Stoica, R. Morris, D. R. Karger, M. F. Kaashoek, H. Balakrishnan: Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. SIGCOMM 2001:149–160.

V. S. Sunderam. PVM: A Framework for Parallel Distributed Computing. *Concurrency – Practice and Experience*, 2(4):315–339, 1990

R. L. Wexelblat (Ed). History of Programming Languages. Academic Press, 1981