Т а б л и ц а 1 — Зависимость химических показателей игристого вина от способа деметаллизации шампанского виноматериала

Показатели	Виноматериал			Игристое вино из виноматериала		
	необработанный	обработанный ЖКС	обработанный электродиализом	необработанного	обработанного ЖКС	обработанного электродиализом
Спирт, % об.	11.0	11.0	11.0	12.1	12.0	12.0
Сахара, г / 100 мл	0.2	0.2	0.2	0.3	0.3	0.3
Титруемая кислотность, г / л	8.6	8.5	8.3	8.7	8.6	8.5
Летучие кислоты, г / л	0.65	0.60	0.54	0.57	0.70	0.63
Общий азот, мг / л	255	234	228	165	168	152
Содержание железа, мг / л	19.2	3.4	3.2	13.2	2.7	2.5
Продолжительность шампанизации, сут.	-	-	-	18	18	18
Показатель игристых свойств (*m)*	-	-	-	2.21	2.35	2.32
Показатель пенистых свойств (*n*)	-	-	-	0.40	0.38	0.43

Заключение. Таким образом, можно считать обоснованным применение метода электродиализа в производстве игристых вин при подготовке виноматериалов перед вторичным брожением. Предлагаемая технология позволяет обеспечить необходимую стабильность готового вина против металлических и кристаллических помутнений без использования таких трудоемких и продолжительных операций как оклейка виноматериалов ЖКС, бентонитом, танином, рыбным клеем, осветление и фильтрация, а также обработка холодом. По качественным показателям игристое вино, полученное из виноматериала, обработанного в электромембранном аппарате, не уступает контрольным образцам, приготовленным по традиционной технологии.

Список цитируемых источников

1. *Кишковский, З. Н.* Технология вина : монография / З. Н. Кишковский, А. А. Мержаниан. — М. : Легкая и пищевая промышленность, 1984. — 504 с.
2. *Исламов, М. Н.* Мембранные технологии в пищевых производствах : монография / М. Н. Исламов. — М. : Академия, 2016. — 213 с.

UDC 378

S. H. Sunnatova
Samarkand State University named after Sharof Rashidov, Samarkand, Republic of Uzbekistan

APPLICATION OF DISTRIBUTED TECHNOLOGIES TO INFORMATION SYSTEMS DEVELOPMENT

Introduction. As the use of digital technologies expands in the development period, the amount of information it contains increases, and its processing and storage becomes more complex. In addition to data storage, increasing the speed and quality of data use is a pressing issue. To increase the speed of use of large amounts of data, there are different approaches. Depending on the characteristics of the system being built, central, distributed, or distributed systems can be developed. No matter how many servers there are in a centralized system, only one server makes decisions. This can be achieved by increasing the number of decision servers and allowing users to choose the servers, creating a decentralized system. Networks of computers are everywhere. The Internet is one, as are the many networks of which it is composed. Mobile phone networks, corporate networks, factory networks, campus networks, home networks, in-car networks — all of these, both separately and in combination, share the essential characteristics that make them relevant subjects for study under the heading distributed systems. Servers in a distributed system have no decision authority. This architecture creates an automatic decision-making system. In this way, data processing is carried out without human intervention, regardless of the content of the data being stored or transmitted. Information in a network is often encrypted. The network participants do not know exactly what information is stored or transmitted. In the thesis, we will try to provide detailed information about the organization of distributed systems, their definition, types, characteristics, advantages, and disadvantages.

A distributed system is a set of two or more computing machines that do not have common memory. But at the same time, they have the ability to work separately and solve problems together.

The ability to work separately increases fault tolerance and efficiently organizes continuous work. A distributed system uses multiple servers with the same memory and software. If one server or device experiences a problem, the other can detect the problem and perform the task independently. Another advantage of a distributed system is its horizontal scalability, which can be scaled to meet the needs of applications and databases. For example, the database may need to be scaled as required when a large amount of data is loaded. A distributed system expands by adding new servers and devices that increase the capacity and performance of the network. Time is an important factor for users and businesses, and a distributed system saves time in performing tasks by using the server closest to the user. Another advantage of this system is that it increases the efficiency and speed of computing functions and processes using multiple servers.

Sharing resources such as hardware, software, and data is one of the principles of cloud computing. With different levels of openness to the software and concurrency, it's easier to process data simultaneously through multiple processors. The more fault-tolerant an application is, the more quickly it can recover from a system failure.

Organizations have turned to distributed computing systems to handle data generation explosion and increased application performance needs. These distributed systems help businesses scale as data volume grows. This is especially true because the process of adding hardware to a distributed system is simpler than upgrading and replacing an entire centralized system made up of powerful servers.

Main part. Distributed systems consist of many nodes that work together toward a single goal. These systems function in two general ways, and both of them have the potential to make a huge difference in an organization.

− The first type is a cohesive system where the customer has each machine, and the results are routed from one source.

− The second type allows each node to have an end-user with their own needs, and the distributed system facilitates sharing resources or communication.

− Benefits of a multi-computer model

− Improved scalability: Distributed computing clusters are a great way to scale your business. They use a 'scale-out architecture,' which makes adding new hardware easier as load increases.

− Enhanced performance: This model uses 'parallelism' for the divide-and-conquer approach. In other words, all computers in the cluster simultaneously handle a subset of the overall task. Therefore, as the load increases, businesses can add more computers and optimize overall performance.

− Cost-effectiveness: The cost-efficiency of a distributed system depends on its latency, response time, bandwidth, and throughput. Distributed systems work toward a common goal of delivering high performance by minimizing latency and enhancing response time and throughput. They achieve this goal by using low-cost commodity hardware to ensure zero data loss, making initial deployments and cluster expansions easy.

We define a distributed system as one in which hardware or software components located at networked computers communicate and coordinate their actions only by passing messages. This simple definition covers the entire range of systems in which networked computers can usefully be deployed.

Computers that are connected by a network may be spatially separated by any distance. They may be on separate continents, in the same building or in the same room. Our definition of distributed systems has the following significant consequences:

Concurrency: In a network of computers, concurrent program execution is the norm. I can do my work on my computer while you do your work on yours, sharing resources such as web pages or files when necessary. The capacity of the system to handle shared resources can be increased by adding more resources (for example. computers) to the network. We will describe ways in which this extra capacity can be usefully deployed at many points in this book. The coordination of concurrently executing programs that share resources is also an important and recurring topic.

No global clock: When programs need to cooperate they coordinate their actions by exchanging messages. Close coordination often depends on a shared idea of the time at which the programs' actions occur. But it turns out that there are limits to the accuracy with which the computers in a network can synchronize their clocks — there is no single global notion of the correct time. This is a direct consequence of the fact that the only communication is by sending messages through a network.

Independent failures: All computer systems can fail, and it is the responsibility of system designers to plan for the consequences of possible failures. Distributed systems can fail in new ways. Faults in the network result in the isolation of the computers that are connected to it, but that doesn't mean that they stop running. In fact, the programs on them may not be able to detect whether the network has failed or has become unusually slow. Similarly, the failure of a computer, or the unexpected termination of a program somewhere in the system (a crash), is not immediately made known to the other components with which it communicates. Each component of the system can fail independently, leaving the others still running.

The prime motivation for constructing and using distributed systems stems from a desire to share resources. The term "resource" is a rather abstract one, but it best characterizes the range of things that can usefully be shared in a networked computer system. It extends from hardware components such as disks and printers to software-defined

entities such as files, databases and data objects of all kinds. It includes the stream of video frames that emerges from a digital video camera and the audio connection that a mobile phone call represents.

The goal of this section is to provide motivational examples of contemporary distributed systems illustrating both the pervasive role of distributed systems and the great diversity of the associated applications. As mentioned in the introduction, networks are everywhere and underpin many everyday services that we now take for granted: the Internet and the associated World Wide Web, web search, online gaming, email, social networks, eCommerce, etc. As can be seen, distributed systems encompass many of the most significant technological developments of recent years and hence an understanding of the underlying technology is absolutely central to a knowledge of modern computing. The figure also provides an initial insight into the wide range of applications in use today, from relatively localized systems (as found, for example, in a car or aircraft) to global-scale systems involving millions of nodes, from data-centric services to processor-intensive tasks, from systems built from very small and relatively primitive sensors to those incorporating powerful computational elements, from embedded systems to ones that support a sophisticated interactive user experience, and so on. We now look at more specific examples of distributed systems to further illustrate the diversity and indeed complexity of distributed systems provision today.

Software architecture is the logical organization of software components and how they interact with other structures. It is at a lower level than system architecture and focuses entirely on components; for example, the web interface of an e-commerce system is a component. The four main architectural styles of distributed systems in software components include.

Layered architecture provides a modular approach to software. It becomes more efficient by separating each component. For example, the open systems interoperability model uses the Layered architecture to achieve better results. The layered architecture is a type of software that divides components into blocks. The request comes from the top down and the response comes from the bottom up. The advantage of the layered architecture is that it organizes everything. Each layer can be changed independently without affecting the rest of the system.

Layered architecture is a type of software that separates components into units. A request goes from the top down, and the response goes from the bottom up. The advantage of layered architecture is that it keeps things orderly and modifies each layer independently without affecting the rest of the system.

Object-based architecture centers around an arrangement of loosely coupled objects with no specific architecture like layers. Unlike layered architecture, object-based architecture doesn't have to follow any steps in a sequence. Each component is an object, and all the objects can interact through an interface (or connector). Under object-based architecture, such interactions between components can happen through a direct method call.

At its core, communication between objects happens through method invocations, often called remote procedure calls (RPC). Popular RPC systems include Java RMI and Web Services and REST API Calls. The primary design consideration of these architectures is that they are less structured. Here, component equals object, and connector equals RPC or RMI.

Data-centered architecture works on a central data repository, either active or passive. Like most producer-consumer scenarios, the producer (business) produces items to the common data store, and the consumer (individual) can request data from it. Sometimes, this central repository can be just a simple database.

All communication between objects happens through a data storage system in a data-centered system. It supports its stores' components with a persistent storage space such as an SQL database, and the system stores all the nodes in this data storage.

In event-based architecture, the entire communication is through events. When an event occurs, the system gets the notification. This means that anyone who receives this event will also be notified and has access to information. Sometimes, these events are data, and at other times they are URLs to resources. As such, the receiver can process what information they receive and act accordingly.

One significant advantage of event-based architecture is that the components are loosely coupled. Eventually, it means that it's easy to add, remove, and modify them. To better understand this, think of publisher-subscriber systems, enterprise services buses, or akka.io. One advantage of event-based architecture is allowing heterogeneous components to communicate with the bus, regardless of their communication protocols.

Distributed systems [1] have the following architectures, depending on their location on the network:

Client-server: This is a self-distributed system that distributes requests between clients and servers. The user sends his requests to the server, and the server assigned to him processes the requests and implements resource management and security policies. An example of a client-server architecture is a web application where the web browser is the client and the web server is the server.

Peer-to-peer: This architecture has no central server or control center. Instead, each node (or peer-to-peer node) in the system can act as a client or server depending on the situation. Nodes are in direct communication with each other, without intermediaries. Nodes are usually autonomous, i. e. they have their resources and functions and can join or leave the system at any time. An example of a node architecture is a file-sharing system where nodes can upload and download files from each other.

If a new node wishes to provide services, it can do so in two ways. One way is to register with a centralized lookup server, which will then direct the node to the service provider. The other way is for the node to broadcast its service request to every other node in the network, and whichever node responds will provide the requested service.

P2P networks of today have three separate sections:

- Structured P2P: The nodes in structured P2P follow a predefined distributed data structure.
- Unstructured P2P: The nodes in unstructured P2P randomly select their neighbors.
- Hybrid P2P: In a hybrid P2P, some nodes have unique functions appointed to them in an orderly manner.

Three-tier: Three-tier distributed systems are client-server architectures organized into three computing tiers: presentation, application, and data. It uses separate servers and tiers for each application function. The presentation tier consists of the user interface and the application tier has the ability to access and process data in the database. The data layer hosts and stores user data and the database. Any change to this system does not require a change to the entire system.

N-tier: The N-tier architecture is essentially similar to the three-tier architecture of distributed systems, but in this (N-tier) architecture each function is performed on a separate machine or cluster of machines.

Examples of a Distributed System

When processing power is scarce, or when a system encounters unpredictable changes, distributed systems are ideal, and they help balance the workload. Hence distributed systems have boundless use cases varying from electronic banking systems to multiplayer online games. Let's check out more explicit instances of distributed systems:

1. Networks.

The 1970s saw the invention of Ethernet and LAN (local area networks), which enabled computers to connect in the same area. Peer-to-peer networks developed, and e-mail and the internet continue to be the biggest examples of distributed systems.

2. Telecommunication networks.

Telephone and cellular networks are other examples of peer-to-peer networks. Telephone networks started as an early example of distributed communication, and cellular networks are also a form of distributed communication systems. With the implementation of Voice over Internet (VoIP) communication systems, they grow more complex as distributed communication networks.

3. Real-time systems.

Real-time systems are not limited to specific industries. These systems can be used and seen throughout the world in the airline, ride-sharing, logistics, financial trading, massively multiplayer online games (MMOGs), and ecommerce industries. The focus in such systems is on the correspondence and processing of information with the need to convey data promptly to a huge number of users who have an expressed interest in such data.

4. Parallel processors.

Parallel computing splits specific tasks among multiple processors. This, in turn, creates pieces to put together and form an extensive computational task. Previously, parallel computing only focused on running software on multiple threads or processors accessing the same data and memory. As operating systems became more prevalent, they too fell into the category of parallel processing.

5. Distributed database systems.

A distributed database is spread out across numerous servers or regions. Data can be replicated across several platforms. A distributed database system can be either homogeneous or heterogeneous in nature. A homogeneous distributed database uses the same database management system and data model across all systems.

Adding new nodes and locations makes it easier to control and scale performance. On the other hand, multiple data models and database management systems are possible with heterogeneous distributed databases. Gateways are used to translate data across nodes and are typically created due to the merger of two or more applications or systems.

6. Distributed artificial intelligence.

Distributed artificial intelligence is one of the many approaches of artificial intelligence that is used for learning and entails complex learning algorithms, large-scale systems, and decision making. It requires a large set of computational data points located in various locations.

A few real-world examples of distributed systems include:

1. Video-rendering systems
2. Scientific computing
3. Airline and hotel reservation
4. Cryptocurrency processors like Bitcoin
5. P2P file-sharing like BitTorrent
6. Multiplayer video games
7. E-learning applications
8. Distributed supply chains like Amazon

Distributed systems [2] are the most significant benefactor behind modern computing systems due to their capability of providing scalable and improved performance. Distributed systems are an essential component of wireless networks, cloud computing, and the internet. Since they can draw on the resources of other devices and processes, distributed systems offer some features that would be hard or even impossible to develop on a singular system and have become immensely reliable by combining the power of multiple machines.

There is a pervasive need for measures to guarantee the privacy, integrity and availability of resources in distributed systems. Security attacks take the forms of eavesdropping, masquerading, tampering and denial of service. Designers of secure distributed systems must cope with exposed service interfaces and insecure networks in an environment where attackers are likely to have knowledge of the algorithms used and to deploy computing resources. Cryptography provides the basis for the authentication of messages as well as their secrecy and integrity; carefully designed security protocols are required to exploit it. The selection of cryptographic algorithms and the management of keys are critical to the effectiveness, performance and usability of security mechanisms. Public-key cryptography makes it easy to distribute cryptographic keys but its performance is inadequate for the encryption of bulk data. Secret-key cryptography is more suitable for bulk encryption tasks. Hybrid protocols such as Transport Layer Security (TLS) establish a secure channel using public-key cryptography and then use it to exchange secret keys for use in subsequent data exchanges. Digital information can be signed, producing digital certificates. Certificates enable trust to be established among users and organizations.

We concluded that the need for security mechanisms in distributed systems arises from the desire to share resources. (Resources that are not shared can generally be protected by isolating them from external access.) If we regard shared resources as objects, then the requirement is to protect any processes that encapsulate shared objects and any communication channels that are used to interact with them against all conceivable forms of attack. The model introduced provides a good starting point for the identification of security requirements. It can be summarized as follows:

Processes encapsulate resources (both programming language–level objects and system-defined resources) and allow clients to access them through interfaces. Principals (users or other processes) are authorized to operate on resources. Resources must be protected against unauthorized access Processes interact through a network that is shared by many users. Enemies (attackers) can access the network. They can copy or attempt to read any message transmitted through the network and they can inject arbitrary messages, addressed to any destination and purporting to come from any source, into the network. The need to protect the integrity and privacy of information and other resources belonging to individuals and organizations is pervasive in both the physical and the digital world. It arises from the desire to share resources. In the physical world, organizations adopt security policies that provide for the sharing of resources within specified limits. For example, a company may permit entry to its buildings only to its employees and accredited visitors. A security policy for documents may specify groups of employees who can access classes of documents, or it may be defined for individual documents and users. Security policies are enforced with the help of security mechanisms. For example, access to a building may be controlled by a reception clerk, who issues badges to accredited visitors, and enforced by a security guard or by electronic door locks. Access to paper documents is usually controlled by concealment and restricted distribution. In the electronic world, the distinction between security policies and mechanisms is equally important, without it, it would be difficult to determine whether a particular system was secure. Security policies are independent of the technology used, just as the provision of a lock on a door does not ensure the security of a building unless there is a policy for its use (for example, that the door will be locked whenever nobody is guarding the entrance). The security mechanisms that we describe here do not in themselves ensure the security of a system.

A distributed file system enables programs to store and access remote files exactly as they do local ones, allowing users to access files from any computer on a network. The performance and reliability experienced for access to files stored at a server should be comparable to that for files stored on local disks. We define a simple architecture for file systems and describe two basic distributed file service implementations with contrasting designs that have been in widespread use for over two decades:

- the Sun Network File System, NFS;
- the Andrew File System, AFS.

Each emulates the UNIX file system interface, with differing degrees of scalability, fault tolerance and deviation from the strict UNIX one-copy file update semantics. Several related file systems that exploit new modes of data organization on disk or across multiple servers to achieve high-performance, fault-tolerant and scalable file systems are also reviewed.

The requirements for sharing within local networks and intranets lead to a need for a different type of service — one that supports the persistent storage of data and programs of all types on behalf of clients and the consistent distribution of up-to-date data. The purpose is to describe the architecture and implementation of these basic distributed file systems. We use the word “basic” here to denote distributed file systems whose primary purpose is to emulate the functionality of a non-distributed file system for client programs running on multiple remote computers. They do not maintain multiple persistent replicas of files, nor do they support the bandwidth and timing guarantees required for multimedia data streaming — those requirements are addressed. Basic distributed file systems provide an essential underpinning for organizational computing based on intranets.

File systems were originally developed for centralized computer systems and desktop computers as an operating system facility providing a convenient programming interface to disk storage. They subsequently acquired features such as access-control and file-locking mechanisms that made them useful for the sharing of data and programs. Distributed file systems support the sharing of information in the form of files and hardware resources in the form of persistent storage throughout an intranet. A well-designed file service provides access to files stored at a server with performance and reliability similar to, and in some cases better than, files stored on local disks. Their

design is adapted to the performance and reliability characteristics of local networks, and hence they are most effective in providing shared persistent storage for use in intranets. The first file servers were developed by researchers in the 1970s [Birrell and Needham 1980, Mitchell and Dion 1982, Leach et al. 1983], and Sun's Network File System became available in the early 1980s [Sandberg et al. 1985, Callaghan 1999].

A file service enables programs to store and access remote files exactly as they do local ones, allowing users to access their files from any computer in an intranet. The concentration of persistent storage at a few servers reduces the need for local disk storage and (more importantly) enables economies to be made in the management and archiving of the persistent data owned by an organization. Other services, such as the name service, the user authentication service and the print service, can be more easily implemented when they can call upon the file service to meet their needs for persistent storage. Web servers are reliant on filing systems for the storage of the web pages that they serve. In organizations that operate web servers for external and internal access via an intranet, the web servers often store and access the material from a local distributed file system.

With the advent of distributed object-oriented programming, a need arose for the persistent storage and distribution of shared objects. One way to achieve this is to serialize objects and to store and retrieve the serialized objects using files. But this method for achieving persistence and distribution is impractical for rapidly changing objects, so several more direct approaches have been developed. Java remote object invocation and CORBA ORBs provide access to remote, shared objects, but neither of these ensures the persistence of the objects, nor are the distributed objects replicated.

The consistency column indicates whether mechanisms exist for the maintenance of consistency between multiple copies of data when updates occur. Virtually all storage systems rely on the use of caching to optimize the performance of programs. Caching was first applied to main memory and non-distributed file systems, and for those the consistency is strict programs cannot observe any discrepancies between cached copies and stored data after an update. When distributed replicas are used, strict consistency is more difficult to achieve. Distributed file systems such as Sun NFS and the Andrew File System cache copies of portions of files at client computers, and they adopt specific consistency mechanisms to maintain an approximation to strict consistency — this is indicated by a tick ( ) in the consistency column.

The Web uses caching extensively both at client computers and at proxy servers maintained by user organizations. The consistency between the copies stored at web proxies and client caches and the original server is only maintained by explicit user actions. Clients are not notified when a page stored at the original server is updated; they must perform explicit checks to keep their local copies up-to-date. This serves the purposes of web browsing adequately, but it does not support the development of cooperative applications such as a shared distributed whiteboard. The consistency mechanisms used in DSM systems are discussed in depth on the companion web site to the book [www.cdk5.net]. Persistent object systems vary considerably in their approach to caching and consistency. The CORBA and Persistent Java schemes maintain single copies of persistent objects, and remote invocation is required to access them, so the only consistency issue is between the persistent copy of an object on disk and the active copy in memory, which is not visible to remote clients. The PerDiS and Khazana projects that we mentioned above maintain cached replicas of objects and employ quite elaborate consistency mechanisms to produce forms of consistency similar to those found in DSM systems.

File systems are responsible for the organization, storage, retrieval, naming, sharing and protection of files. They provide a programming interface that characterizes the file abstraction, freeing programmers from concern with the details of storage allocation and layout. Files are stored on disks or other non-volatile storage media. Files contain both data and attributes. The data consist of a sequence of data items (typically 8-bit bytes), accessible by operations to read and write any portion of the sequence. The attributes are held as a single record containing information such as the length of the file, timestamps, file type, owner's identity and access control lists. A typical attribute record structure is illustrated. The shaded attributes are managed by the file system and are not normally updatable by user programs.

File systems are designed to store and manage large numbers of files, with facilities for creating, naming and deleting files. The naming of files is supported by the use of directories. A directory is a file, often of a special type, that provides a mapping from text names to internal file identifiers. Directories may include the names of other directories, leading to the familiar hierarchic file-naming scheme and the multi-part pathnames for files used in UNIX and other operating systems. File systems also take responsibility for the control of access to files, restricting access to files according to users' authorizations and the type of access requested (reading, updating, executing and so on). The term metadata is often used to refer to all of the extra information stored by a file system that is needed for the management of files. It includes file attributes, directories and all the other persistent information used by the file system.

File system operations. The main operations on files that are available to applications in UNIX systems. These are the system calls implemented by the kernel; application programmers usually access them through procedure libraries such as the C Standard Input/Output Library or the Java file classes.

The UNIX operations are based on a programming model in which some file state information is stored by the file system for each running program. This consists of a list of currently open files with a read-write pointer for each, giving the position within the file at which the next read or write operation will be applied.

The file system is responsible for applying access control for files. In local file systems such as UNIX, it does so when each file is opened, checking the rights allowed for the user's identity in the access control list against the mode of access requested in the open system call. If the rights match the mode, the file is opened and the mode is recorded in the open file state information.

Distributed file system requirements

Many of the requirements and potential pitfalls in the design of distributed services were first observed in the early development of distributed file systems. Initially, they offered access transparency and location transparency; performance, scalability, concurrency control, fault tolerance and security requirements emerged and were met in subsequent phases of development. We discuss these and related requirements in the following subsections.

Transparency • The file service is usually the most heavily loaded service in an intranet, so its functionality and performance are critical. The design of the file service should support many of the transparency requirements for distributed systems identified. The design must balance the flexibility and scalability that derive from transparency against software complexity and performance. The following forms of transparency are partially or wholly addressed by current file services:

Access transparency: Client programs should be unaware of the distribution of files. A single set of operations is provided for access to local and remote files. Programs written to operate on local files are able to access remote files without modification. Location transparency: Client programs should see a uniform file name space. Files or groups of files may be relocated without changing their pathnames, and user programs see the same name space wherever they are executed.

Mobility transparency: Neither client programs nor system administration tables in client nodes need to be changed when files are moved. This allows file mobility — files or, more commonly, sets or volumes of files may be moved, either by system administrators or automatically.

Performance transparency: Client programs should continue to perform satisfacto-rily while the load on the service varies within a specified range.

Scaling transparency: The service can be expanded by incremental growth to deal with a wide range of loads and network sizes.

Concurrent file updates • Changes to a file by one client should not interfere with the operation of other clients simultaneously accessing or changing the same file. This is the well-known issue of concurrency control, discussed in detail. The need for concurrency control for access to shared data in many applications is widely accepted and techniques are known for its implementation, but they are costly. Most current file services follow modern UNIX standards in providing advisory or mandatory file- or record-level locking.

File replication • In a file service that supports replication, a file may be represented by several copies of its contents at different locations. This has two benefits — it enables multiple servers to share the load of providing a service to clients accessing the same set of files, enhancing the scalability of the service, and it enhances fault tolerance by enabling clients to locate another server that holds a copy of the file when one has failed. Few file services support replication fully, but most support the caching of files or portions of files locally, a limited form of replication. Hardware and operating system heterogeneity • The service interfaces should be de-fined so that client and server software can be implemented for different operating sys-tems and computers. This requirement is an important aspect of openness.

Fault tolerance • The central role of the file service in distributed systems makes it essential that the service continue to operate in the face of client and server failures. Fortunately, a moderately fault-tolerant design is straightforward for simple servers. To cope with transient communication failures, the design can be based on at-most-once invocation or it can use the simpler at-least-oncesemantics with a server protocol designed in terms of idempotent operations, ensuring that duplicated requests do not result in invalid updates to files. The servers can be stateless, so that they can be restarted and the service restored after a failure without any need to recover previous state. Tolerance of disconnection or server failures requires file replication, which is more difficult to achieved.

Consistency • Conventional file systems such as that provided in UNIX offer one-copy update semantics. This refers to a model for concurrent access to files in which the file contents seen by all of the processes accessing or updating a given file are those that they would see if only a single copy of the file contents existed. When files are replicated or cached at different sites, there is an inevitable delay in the propagation of modifications made at one site to all of the other sites that hold copies, and this may result in some deviation from one-copy semantics.

Security • Virtually all file systems provide access-control mechanisms based on the use of access control lists. In distributed file systems, there is a need to authenticate client requests so that access control at the server is based on correct user identities and to protect the contents of request and reply messages with digital signatures and (optionally) encryption of secret data. We discuss the impact of these requirements in the case studies later.

Efficiency • A distributed file service should offer facilities that are of at least the same power and generality as those found in conventional file systems and should achieve a comparable level of performance. Birrell and Needham [1980] expressed their design aims for the Cambridge File Server (CFS) in these terms:

We would wish to have a simple, low-level file server in order to share an expensive resource, namely a disk, whilst leaving us free to design the filing system most appropriate to a particular client, but we would wish also to have available a high-level system shared between clients. The changed economics of disk storage have reduced the

significance of their first goal, but their perception of the need for a range of services addressing the requirements of clients with different goals remains and can best be addressed by a modular architecture of the type outlined above.

Conclusion. The techniques used for the implementation of file services are an important part of the design of distributed systems. A distributed file system should provide a service that is comparable with, or better than, local file systems in performance and reliability. It must be convenient to administer, providing operations and tools that enable system administrators to install and operate the system conveniently.

We have constructed an abstract model for a file service to act as an introductory example, separating implementation concerns and providing a simplified model. We describe the Sun Network File System in some detail, drawing on our simpler abstract model to clarify its architecture. The Andrew File System is then described, providing a view of a distributed file system that takes a different approach to scalability and consistency maintenance.

File service architecture • This is an abstract architectural model that underpins both NFS and AFS. It is based upon a division of responsibilities between three modules — a client module that emulates a conventional file system interface for application programs, and server modules, that perform operations for clients on directories and on files. The architecture is designed to enable a stateless implementation of the server module.

SUN NFS • Sun Microsystems's Network File System (NFS) has been widely adopted in industry and in academic environments since its introduction in 1985. The design and development of NFS were undertaken by staff at Sun Microsystems in 1984 [Sandberg et al. 1985, Sandberg 1987, Callaghan 1999]. Although several distributed file services had already been developed and used in universities and research laboratories, NFS was the first file service that was designed as a product. The design and implementation of NFS have achieved success both technically and commercially.

List of cited sources

1. Distributed Systems : Concepts and Design Fifth Edition / G. Coulouris [et al.]. — United States of America : Inc., publishing as Addison-Wesley. 2012 — 1067 p.

2. Distributed Systems Architecture / San Francisco : Elsevier. 2006 — 342 p.