Using Abstraction To Improve Fault Tolerance

Miguel Castro
Microsoft Research Ltd.
1 Guildhall St., Cambridge CB2 3NH, UK
mcastro@microsoft.com

Rodrigo Rodrigues and Barbara Liskov
MIT Laboratory for Computer Science
545 Technology Sq., Cambridge, MA 02139, USA
{rodrigo,liskov}@lcs.mit.edu

Abstract

Software errors are a major cause of outages and they are increasingly exploited in malicious attacks. Byzantine fault tolerance allows replicated systems to mask some software errors but it is expensive to deploy. This paper describes a replication technique, BFTA, which uses abstraction to reduce the cost of Byzantine fault tolerance and to improve its ability to mask software errors. BFTA reduces cost because it enables reuse of off-the-shelf service implementations. It improves availability because each replica can be repaired periodically using an abstract view of the state stored by correct replicas, and because each replica can run distinct or non-deterministic service implementations, which reduces the probability of common mode failures. We built an NFS service that allows each replica to run a different operating system. This example suggests that BFTA can be used in practice — the replicated file system required only a modest amount of new code, and preliminary performance results indicate that it performs comparably to the off-the-shelf implementations that it wraps.

1. Introduction

There is a growing demand for highly-available systems that provide correct service without interruptions. These systems must tolerate software errors because these are a major cause of outages [7]. Furthermore, there is an increasing number of malicious attacks that exploit software errors to gain control or deny access to systems that provide important services.

This paper proposes a replication technique, BFTA, that combines Byzantine fault tolerance [12] with work on data abstraction [11]. Byzantine fault tolerance allows a replicated service to tolerate arbitrary behavior from faulty replicas, e.g., the behavior caused by a software bug, or the behavior of a replica that is controlled by an attacker. Abstraction hides implementation details to enable the reuse of off-the-shelf implementations of important services (e.g., file systems, databases, or HTTP daemons) and to improve the ability to mask software errors.

We extended the BFT library [1, 2] to implement BFTA. The original BFT library provides Byzantine fault tolerance with good performance and strong correctness guarantees if no more than $1/3$ of the replicas fail within a small window of vulnerability. However, it requires all replicas to run the same service implementation and to update their state in a deterministic way. Therefore, it cannot tolerate deterministic software errors that cause all replicas to fail concurrently and it complicates reuse of existing service implementations because it requires extensive modifications to ensure identical values for the state of each replica.

The BFTA library and methodology described in this paper correct these problems — they enable replicas to run different or non-deterministic implementations. The methodology is based on the concepts of *abstract specification* and *abstraction function* from work on data abstraction [11]. We start by defining a *common abstract specification* for the service, which specifies an *abstract state* and describes how each operation manipulates the state. Then we implement a *conformance wrapper* for each distinct implementation to make it behave according to the common specification. The last step is to implement an abstraction function (and one of its inverses) to map from the concrete state of each implementation to the common abstract state (and vice versa).

Our methodology offers several important advantages.
Reuse of existing code. BFTA implements a form of state machine replication [14, 10], which allows replication of services that perform arbitrary computations, but requires determinism: all replicas must produce the same sequence of results when they process the same sequence of operations. Most off-the-shelf implementations of services fail to satisfy this condition. For example, many implementations produce timestamps by reading local clocks, which can cause the states of replicas to diverge. The conformance wrapper and the abstract state conversions enable the reuse of existing implementations without modifications. Furthermore, these implementations can be non-deterministic, which reduces the probability of common mode failures.
Software rejuvenation. It has been observed [9] that there

This research was partially supported by DARPA under contract F30602-98-1-0237 monitored by the Air Force Research Laboratory.

is a correlation between the length of time software runs and the probability that it fails. BFTA combines proactive recovery [2] with abstraction to counter this problem. Replicas are recovered periodically even if there is no reason to suspect they are faulty. Recoveries are staggered such that the service remains available during rejuvenation to enable frequent recoveries. When a replica is recovered, it is rebooted and restarted from a clean state. Then it is brought up to date using a correct copy of the abstract state that is obtained from the group of replicas. Abstraction may improve availability by hiding corrupt concrete states, and it enables proactive recovery when replicas do not run the same code or run code that is non-deterministic.

Opportunistic N-version programming. Replication is not useful when there is a strong positive correlation between the failure probabilities of the different replicas, e.g., deterministic software bugs cause all replicas to fail at the same time when they run the same code. BFTA enables an opportunistic form of N-version programming [3] — replicas can run distinct, off-the-shelf implementations of the service. This is a viable option for many common services, e.g., relational databases, HTTP daemons, file systems, and operating systems. In all these cases, competition has led to four or more distinct implementations that were developed and are maintained separately but have similar (although not identical) functionality. Furthermore, the technique is made easier by the existence of standards that provide identical interfaces to different implementations, e.g., ODBC [6] and NFS [5]. We can also leverage the large effort towards standardizing data representations using XML.

It is widely believed that the benefits of N-version programming [3] do not justify its high cost [7]. It is better to invest the same amount of money on better development, verification, and testing of a single implementation. But opportunistic N-version programming achieves low cost due to economies of scale without compromising the quality of individual implementations. Since each off-the-shelf implementation is sold to a large number of customers, the vendors can amortize the cost of producing a high quality implementation. Additionally, taking advantage of interoperability standards keeps the cost of writing the conformance wrappers and state conversion functions low.

The paper explains the methodology by walking through an example, the implementation of a replicated file service where replicas run different operating systems and file systems. For this methodology to be successful, the conformance wrapper and the state conversion functions must be simple to reduce the likelihood of introducing more errors and introduce a low overhead. Experimental results indicate that this is true in our example.

The remainder of the paper is organized as follows. Section 2 provides an overview of the BFTA methodology and library. Section 3 explains how we applied the methodology to build the replicated file system. Section 4 presents our conclusions and some preliminary results.

2. The BFTA Technique

This section provides an overview of our replication technique. It starts by describing the methodology that we use to build a replicated system from existing service implementations. It ends with a description of the BFTA library.

2.1. Methodology

The goal is to build a replicated system by reusing a set of off-the-shelf implementations, $I_1, ..., I_n$, of some service. Ideally, we would like n to equal the number of replicas so that each replica can run a different implementation to reduce the probability of simultaneous failures. But the technique is useful even with a single implementation.

Although off-the-shelf implementations of the same service offer roughly the same functionality, they behave differently: they implement different specifications, $S_1, ..., S_n$ using different representations of the service state. Even the behavior of different replicas that run the same implementation may be different when the specification they implement is not strong enough to ensure deterministic behavior. For instance, the specification of the NFS protocol [5] allows implementations to choose arbitrary values for file handles.

BFTA, like any form of state machine replication, requires determinism: replicas must produce the same sequence of results when they execute the same sequence of operations. We achieve determinism by defining a *common abstract specification*, S, for the service that is strong enough to ensure deterministic behavior. This specification defines the abstract state, an initial state value, and the behavior of each service operation.

The specification is defined without knowledge of the internals of each implementation unlike what happens in the technique sketched in [13]. It is sufficient to treat them as black boxes, which is important to enable the use of existing implementations. Additionally, the abstract state captures only what is visible to the client rather than mimicking what is common in the concrete states of the different implementations. This simplifies the abstract state and improves the effectiveness of our software rejuvenation technique.

The next step, is to implement *conformance wrappers*, $C_1, ..., C_n$, for each of $I_1, ..., I_n$. The conformance wrappers implement the common specification S. The implementation of each wrapper C_i is a veneer that invokes the operations offered by I_i to implement the operations in S; in implementing these operations it makes use of a *conformance rep* that stores whatever additional information is needed to allow the translation from the concrete behavior of the implementation to the abstract behavior.

The final step is to implement the *abstraction function* and one of its inverses. These functions allow state transfer among the replicas. State transfer is used to repair faulty replicas, and also to bring slow replicas up-to-date when

messages they are missing have been garbage collected. For state transfer to work replicas must agree on the value of the state of the service after executing a sequence of operations; they will not agree on the value of the concrete state but our methodology ensures that they will agree on the value of the abstract state. The abstraction function is used to convert the concrete state stored by a replica into the abstract state, which is transferred to another replica. The receiving replica uses the inverse function to convert the abstract state into its own concrete state representation.

To enable efficient state transfer between replicas, the abstract state is defined as an array of variable-sized objects. We explain how this representation enables efficient state transfer in Section 2.2.

There is an important trend that simplifies the methodology. Market forces push vendors towards extending their products to offer interfaces that implement standard specifications for interoperability, e.g., ODBC [6]. Usually, a standard specification S' cannot be used as the common specification S because it is too weak to ensure deterministic behavior. But it can be used as a basis for S and, because S and S' are similar, it is relatively easy to implement conformance wrappers and state conversion functions, these implementations can be mostly reused across implementations, and most client code can use the replicated system without modification.

2.2. Library

The BFTA library extends BFT with the features necessary to provide the methodology. Figure 1 presents a summary of the library's interface.

```
Client call:
int invoke(Byz_req *req, Byz_rep *rep,
                              bool read_only);

Execution upcall:
int execute(Byz_req*req, Byz_rep*rep,
            int client, Byz_buffer *non-det);

Checkpointing:
void modify(int nobjs, int* objs);

State conversion upcalls:
int get_obj(int i, char** obj);

void put_objs(int nobjs, char **objs,
                           int *is, int *szs);
```

Figure 1. BFTA Interface and Upcalls

The `invoke` procedure is called by the client to invoke an operation on the replicated service. This procedure carries out the client side of the replication protocol and returns the result when enough replicas have responded. When the library needs to execute an operation at a replica, it makes an upcall to an `execute` procedure that is implemented by the conformance wrapper for the service implementation run by the replica.

To perform state transfer in the presence of Byzantine faults, it is necessary to be able to prove that the state being transferred is correct. Otherwise, faulty replicas could corrupt the state of out-of-date but correct replicas. (A detailed discussion of this point can be found in [2].) Consequently, replicas cannot discard a copy of the state produced after executing a request until they know that the state produced by executing later requests can be proven correct. Replicas could keep a copy of the state after executing each request but this would be too expensive. Instead replicas keep just the current version of the concrete state plus copies of the abstract state produced every k-th request (e.g., k=128). These copies are called checkpoints.

As mentioned earlier, to implement checkpointing and state transfer efficiently, we require that the abstract state be encoded as an array of objects. Creating checkpoints by making full copies of the abstract state would be too expensive. Instead, the library uses copy-on-write such that checkpoints only contain the objects whose value is different in the current abstract state. Similarly, transferring a complete checkpoint to bring a recovering or out-of-date replica up to date would be too expensive. The library employs a hierarchical state partition scheme to transfer state efficiently. When a replica is fetching state, it recurses down a hierarchy of meta-data to determine which partitions are out of date. When it reaches the leaves of the hierarchy (which are the abstract objects), it fetches only the objects that are corrupt or out of date.

To implement state transfer, each replica must provide the library with two upcalls, which implement the *abstraction function* and one of its inverses. `get_obj` receives an object index i, allocates a buffer, obtains the value of the abstract object with index i, and places that value in the buffer. It returns the size for that object and a pointer to the buffer. `put_objs` receives a vector of objects with the corresponding indices and sizes. It causes the application to update its concrete state using the new values for the abstract objects passed as arguments. The library guarantees that the `put_objs` upcall is invoked with an argument that brings the abstract state of the replica to a consistent value (i.e., the value of a valid checkpoint). This is important to allow encodings of the abstract state with dependencies between objects, e.g., it allows objects to describe the meaning of other objects.

Each time the `execute` upcall is about to modify an object in the abstract state it is required to invoke a `modify` procedure, which is supplied by the library, passing the object index as argument. This is used to implement copy-on-write to create checkpoints incrementally: the library invokes `get_obj` with the appropriate index and keeps the copy of the object until the corresponding checkpoint can be discarded.

BFTA implements a form of state machine replication that requires replicas to behave deterministically. The methodology uses abstraction to hide most of the non-determinism

in the implementations it reuses. However, many services involve forms of non-determinism that cannot be hidden by abstraction. For instance, in the case of the NFS service, the time-last-modified for each file is set by reading the server's local clock. If this were done independently at each replica, the states of the replicas would diverge. The library provides a mechanism [1] for replicas to agree on these non-deterministic values, which are then passed as arguments to the `execute` procedure.

Proactive recovery periodically restarts each replica from a correct, up-to-date checkpoint of the abstract state that is obtained from the other replicas. Recoveries are staggered so that less than $1/3$ of the replicas recover at the same time. This allows the other replicas to continue processing client requests during the recovery. Additionally, it should reduce the likelihood of simultaneous failures due to aging problems because at any instant less than $1/3$ of the replicas have been running for the same period of time.

Recoveries are triggered by a watchdog timer. When a replica is recovered, it reboots after saving the replication protocol state and the concrete service state to disk. The protocol state includes the abstract objects that were copied by the incremental checkpointing mechanism. Then the replica is restarted, and the conformance rep is reconstructed using the information that was saved to disk. Next, the library uses the hierarchical state transfer mechanism to compare the value of the abstract state it currently stores with the abstract state values stored by the other replicas. This is efficient: the replica uses cryptographic hashes stored in the state partition tree to determine which abstract objects are out-of-date or corrupt and it only fetches the value of these objects.

The object values fetched by the replica could be supplied to `put_objs` to update the concrete state, but the concrete state might still be corrupt. For example, an implementation may have a memory leak and simply calling `put_objs` will not free unreferenced memory. In fact, implementations will not typically offer an interface that can be used to fix all corrupt data structures in their concrete state. Therefore, it is better to restart the implementation from a clean initial concrete state and use the abstract state to bring it up-to-date.

3. An example: File System

This section illustrates the methodology using a replicated file system as an example. The file system is based on the NFS protocol [5]. Its replicas can run different operating systems and file system implementations.

3.1. Abstract Specification

The common abstract specification is based on the specification of the NFS protocol [5]. The abstract file service state consists of a fixed-size array of pairs with an object and a generation number. Each object has a unique identifier, *oid*, which is obtained by concatenating its index in the array and its generation number. The generation number is incremented every time the entry is assigned to a new object. There are four types of objects: files, whose data is a byte array; directories, whose data is a sequence of <name, oid> pairs ordered lexicographically; symbolic links, whose data is a small character string; and special *null* objects, which indicate an entry is free. All non-null objects have meta-data, which includes the attributes in the NFS `fattr` structure. Each entry in the array is encoded using XDR [4]. The object with index 0 is a directory object that corresponds to the root of the file system tree that was mounted.

The operations in the common specification are those defined by the NFS protocol. There are operations to read and write each type of non-null object. The file handles used by the clients are the *oids* of the corresponding objects. To ensure deterministic behavior, we define a deterministic procedure to assign *oid*s, and require that directory entries returned to a client be ordered lexicographically.

The abstraction hides many details; the allocation of file blocks, the representation of large files and directories, and the persistent storage medium and how it is accessed. This is desirable for simplicity, performance, and to improve resilience to software faults due to aging.

3.2. Conformance Wrapper

The conformance wrapper for the file service processes NFS protocol operations and interacts with an off-the-shelf file system implementation also using the NFS protocol as illustrated in Figure 2. A file system exported by the replicated file service is mounted on the client machine like any regular NFS file system. Application processes run unmodified and interact with the mounted file system through the NFS client in the kernel. We rely on user level relay processes to mediate communication between the standard NFS client and the replicas. A relay receives NFS protocol requests, calls the `invoke` procedure of our replication library, and sends the result back to the NFS client. The replication library invokes the `execute` procedure implemented by the conformance wrapper to run each NFS request.

The conformance rep consists of an array that corresponds to the one in the abstract state but it does not store copies of the objects; instead each array entry contains the generation number, the file handle assigned to the object by the underlying NFS server, and the value of the timestamps in the object's abstract meta-data. Empty entries store a null file handle. The rep also contains a map from file handles to *oids* to aid in processing replies efficiently.

The wrapper processes each NFS request received from a client as follows. It translates the file handles in the request, which encode *oid*s, into the corresponding NFS server file

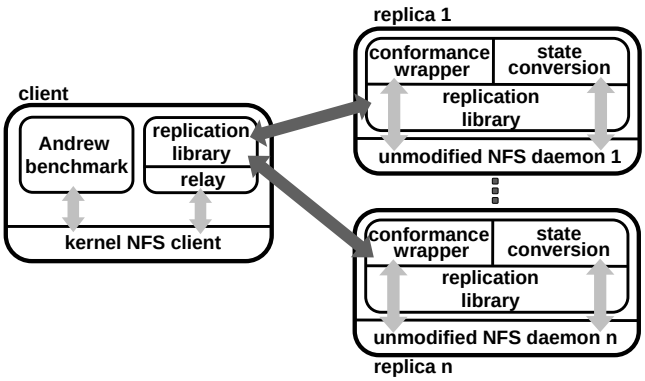

Figure 2. Software Architecture

handles. Then it sends the modified request to the underlying NFS server. The server processes the request and returns a reply.

The wrapper parses the reply and updates the conformance rep. If the operation created a new object, the wrapper allocates a new entry in the array in the conformance rep, increments the generation number, and updates the entry to contain the file handle assigned to the object by the NFS server. If any object is deleted, the wrapper marks its entry in the array free. In both cases, the reverse map from file handles to *oids* is updated. The wrapper must also update the abstract timestamps in the array entries corresponding to objects that were accessed. We use the library to agree on the timestamp value that is assigned to each operation [1]. This value is one of the arguments to the `execute` procedure implemented by the wrapper.

Finally, the wrapper returns a modified reply to the client, using the map to translate file handles to *oids* and replacing the concrete timestamp values by the abstract ones. When handling *readdir* calls the wrapper reads the entire directory and sorts it lexicographically to ensure the client receives identical replies from all replicas.

3.3. State Conversions

The abstraction function in the file service is implemented as follows. For each file system object, it uses the file handle stored in the conformance rep to invoke the NFS server to obtain the data and meta-data for the object. Then it replaces the concrete timestamp values by the abstract ones, converts the file handles in directory entries to *oids*, and sorts the directories lexicographically.

The inverse abstraction function in the file service works as follows. For each file system object o it receives, there are three possible cases depending on the state of the entry e that corresponds to o in the conformance rep: (1) e contains o's generation number, (2) e is not free and does not contain o's generation number, (3) e is free.

In the first case, objects that changed can be updated using the file handle in e to make calls to the NFS server. This is done differently for different types of objects. For files, it is sufficient to issue a `setattr` and a `write` to update the file's meta-data and data, and for symbolic links, it is sufficient to update their meta-data. Updating directories is slightly trickier. The inverse abstraction function reads the entire directory from the NFS server, computes its current abstract value, and compares this value with o. Nothing is done for entries that did not change. Entries that are not present in o or point to a different object are removed by issuing the appropriate calls to the NFS server. Then entries that are new or different in o are created but if the object they refer to does not exist in the current abstract state, it is first created using the value for the object that is supplied to `put_objs`.

In the second case, the NFS server is invoked to remove the object and then the function proceeds as in case 3.

In the third case, the NFS server is invoked to create the object (initially in a separate *unlinked* directory) and the object's data and meta-data is updated as in case 1. It is guaranteed that the directories that point to the object will be processed; the object is then linked to those directories and removed from the unlinked directory. When new objects are created, their file handles are recorded in the conformance wrapper's data structures.

3.4. Proactive Recovery

NFS file handles are volatile: the same file system object may have a different file handle after the NFS server restarts. For proactive recovery to work efficiently, we need a persistent identifier for objects in the concrete file system state that can be used to compute the abstraction function during recovery.

The NFS specification states that each object is uniquely identified by a pair of meta-data attributes: <fsid,fileid>. We solve the problem above by maintaining an additional map from <fsid,fileid> pairs to the corresponding *oids*. This map is saved to disk asynchronously when a checkpoint is created and synchronously before a proactive recovery. After rebooting, the replica that is recovering reads the map from disk. Then it traverses the file system's directory tree depth first from the root. It reads each object, uses the map to obtain its *oid*, and uses the cryptographic hashes from the state transfer protocol to check if the object is up-to-date. If the object is out-of-date or corrupt, it is fetched from another replica.

Instead of simply calling `put_objs` with the new object values, we intend to start an NFS server on a second empty disk and bring it up-to-date incrementally as we obtain the value of the abstract objects. This has the advantage of improving fault-tolerance as discussed in Section 2.2. Additionally, it can improve disk locality by clustering blocks from the same file and files that are in the same directory. This is not done in the current prototype.

4. Conclusion

Software errors are a major cause of outages and they are increasingly exploited in malicious attacks to gain control or deny access to important services. Byzantine fault tolerance allows replicated systems to mask some software errors but it has been expensive to deploy. We have described a replication technique, BFTA, which uses abstraction to reduce the cost of deploying Byzantine fault tolerance and to improve its ability to mask software errors.

BFTA reduces cost because it enables reuse of off-the-shelf service implementations without modifications, and it improves resilience to software errors by enabling opportunistic N-version programming, and software rejuvenation through proactive recovery.

Opportunistic N-version programming runs distinct, off-the-shelf implementations at each replica to reduce the probability of common mode failures. To apply this technique, it is necessary to define a common abstract behavioral specification for the service and to implement appropriate conversion functions for the state, requests, and replies of each implementation in order to make it behave according to the common specification. These tasks are greatly simplified by basing the common specification on standards for the interoperability of software from different vendors; these standards appear to be common, e.g., ODBC [6], and NFS [5]. Opportunistic N-version programming improves on previous N-version programming techniques by avoiding the high development, testing, and maintenance costs without compromising the quality of individual versions.

Additionally, we provide a mechanism to repair faulty replicas. Proactive recovery allows the system to remain available provided no more than $1/3$ of the replicas become faulty and corrupt the abstract state (in a correlated way) within a window of vulnerability. Abstraction may enable more than $1/3$ of the replicas to be faulty because it can hide corrupt items in concrete states of faulty replicas.

The paper described a replicated NFS file system implemented using our technique. The conformance wrapper and the state conversion functions in our prototype are simple — they have 1105 semi-colons, which is two orders of magnitude less than the size of the Linux 2.2 kernel. This suggests that they are unlikely to introduce new bugs.

We ran a scaled-up version of the Andrew benchmark [8, 2] (which generates 1 GB of data) to compare the performance of our replicated file system and the off-the-shelf implementation of NFS in Linux 2.2 that it wraps. Our performance results indicate that the overhead introduced by our technique is low; it is approximately 30% for this benchmark with a window of vulnerability of 17 minutes.

These preliminary results suggest that BFTA can be used in practice. As future work, it would be important to run experiments that apply BFTA to more challenging services, e.g., a relational database. It would also be important to run fault injection experiments to evaluate the availability improvements afforded by our technique.

References

[1] M. Castro and B. Liskov. Practical Byzantine Fault Tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, New Orleans, LA, February 1999.

[2] M. Castro and B. Liskov. Proactive Recovery in a Byzantine-Fault-Tolerant System. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation*, San Diego, CA, October 2000.

[3] L. Chen and A. Avizienis. N-Version Programming: A Fault-Tolerance Approach to Reliability of Software Operation. In *Fault Tolerant Computing, FTCS-8*, pages 3–9, 1978.

[4] Network Working Group Request for Comments: 1014. XDR: External Data Representation Standard, June 1987.

[5] Network Working Group Request for Comments: 1094. NFS: Network File System Protocol Specification, March 1989.

[6] Kyle Geiger. *Inside ODBC*. Microsoft Press, 1995.

[7] J. Gray and D. Siewiorek. High-Availability Computer Systems. *IEEE Computer*, 24(9):39–48, September 1991.

[8] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.

[9] Y. Huang, C. Kintala, N. Kolettis, and N. D. Fulton. Software Rejuvenation: Analysis, Module and Applications. In *Fault-Tolerant Computing, FTCS-25*, pages 381–390, Pasadena, CA, June 1995.

[10] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.

[11] B. Liskov and J. Guttag. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*. Addison-Wesley, 2000.

[12] M. Pease, R. Shostak, and L. Lamport. Reaching Agreement in the Presence of Faults. *Journal of the ACM*, 27(2):228–234, April 1980.

[13] A. Romanovsky. Abstract Object State and Version Recovery in N-Version Programming. In *TOOLS Europe'99*, Nancy, France, June 1999.

[14] F. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.