The Role of Accountability in Dependable Distributed Systems

Aydan R. Yumerefendi and Jeffrey S. Chase
Computer Science Department
Duke University
Durham, NC 27701, USA
aydan,chase@cs.duke.edu

Abstract

This paper promotes accountability as a first-class design principle for dependable network systems. Conventional techniques for dependable systems design are insufficient to defend against an adversary that manipulates the system covertly in order to lie, cheat, or steal. This paper treats subversion as a form of fault, and suggests that designing accountability into the system provides the means to detect, isolate, and tolerate such faults, and even to prevent them by removing incentives for malicious behavior. A key challenge for the future is to extend the repertoire of dependable systems design and analysis with broadly applicable techniques to build systems with target levels of accountability quantified by the probability that an attacker will be exposed.

1. Introduction

Distributed systems today face a broader array of faults and threats than they have in the past. These threats demand new ways of thinking about dependability, and new methodologies to build and evaluate dependable systems.

Networked systems show increasing complexity, coordination, and scale. Their components are often separately developed and evolving independently. They may span trust domains. Most importantly, today's systems often control processes or assets in the real world, and their components may act on behalf of principals with competing interests. This situation creates both the means and the motive to subvert one or more components of the system for nefarious ends: lying, cheating, or stealing.

The threat of attack or subversion is older than computing. The incentives for such behavior are increasingly more evident as new uses of technology allow sharing and integration of attractive resources and functionality among diverse and autonomous groups of users, e.g., peer-to-peer systems, resource-sharing grids, electronic commerce, and utility services. Lying, cheating, and stealing offer system actors (users and components) opportunities to benefit by violating accepted conventions: a lying actor can impugn the reputation of another, a cheating component may receive compensation without delivering a negotiated service, or a thief may consume resources that it should not receive.

It is tempting to view these challenges as belonging to the domain of secure computing rather than dependability. Conventional security technologies can establish a perimeter defense against attacks, but there is no common perimeter to defend when functionality is integrated across trust domains. In any case, once a perimeter is breached the attack becomes a fault that will cause the system to fail, possibly in insidious ways. When trust failures occur it is important to trace the chain of actions and pinpoint the responsible entity, which may behave deceptively.

This particular class of faults requires new solutions, specifically designed to deal with the problems arising from the lack of centralized control and dependence on fragile trust relationships. We argue to address this challenge by elevating *accountability* as a first-class design principle for dependable distributed systems. Accountability has recently been viewed as a desirable property of dependable systems. In this paper, we offer a different perspective on accountability as a *means to an end* of preventing, detecting, isolating, tolerating, and repairing trust faults.

Accountability makes it possible to detect and isolate misbehavior, contain damage, and assign nonrepudiable responsibility for incorrect or inconsistent states and actions [23]. Accountable design complements more general other approaches to building dependable, trustworthy systems, such as secure hardware [20], Byzantine Fault Tolerance based on state-machine replication [19, 7] , or N-version programming [3]. However, accountable design has a greater impact on the structure of dependable components and their and interaction protocols.

Our goal is to promote a broader understanding of accountability and the techniques to provide it, measure it, and build on it to address the threats and challenges arising

from coordination and integration between different trust domains. We also frame some key research issues for accountability in dependable systems.

2. A Brief History

The notion of dependability continues to evolve to respond to changing requirements and changing contexts. Early systems had short, often unacceptable, time to failure [12], so uptime was a primary concern. As reliability improved through better engineering and component replication, focus shifted to overall availability of a service to complete user requests (e.g., [22]) or serve requests at some target level of performance. The scope broadened to encompass safety as systems began to take on real-world functions previously performed by people. Improvements in network access and connectivity exposed systems to a whole world of potential attackers, heightening concerns about unauthorized access, intrusion, and subversion.

The world has changed again. Recent advances in service-oriented computing and scalable networked systems (e.g., P2P) promote integration of functionality across different trust domains. At the same time, we see an increasing dependence on interconnected mission-critical systems in domains including commerce, health care, finance, and government, where actions taken by (or through) software components have significant consequences in the real world. These factors introduce a new requirement for dependable systems: they must defend against "trust faults"—lying, cheating, or stealing by one or more components.

The classical security-centered approach to dealing with such malicious behavior is to *prevent* it by imposing a secure perimeter based on the "gold standard" of authentication and authorization [14]. Unfortunately, perimeter defenses are brittle. Because they are cumbersome and inconvenient, they are deployed inconsistently and are often inadequate to ward off an attacker. Once penetrated, they are worthless. More fundamentally, perimeter defenses offer no protection when an authorized actor requests a valid action in error or with malice. When components interact across trust domains each component has its own perimeter, which cannot defend against subversion of its peers.

Byzantine fault tolerance is a complementary technique that protects the system by detecting and isolating compromised peers. BFT replicates components and uses quorum voting to identify and suppress incorrect actions [5, 7, 19]. This general and well-developed approach is effective when failures are truly independent, but it is vulnerable to collusion and common-mode failures. Moreover, BFT has little value when the component (and hence the replica set) is itself acting on behalf of a malicious entity that controls its behavior. To illustrate this point, consider a malicious managed auction service. To ensure correct auction outcomes, Byzantine fault tolerance would require that each auction takes place simultaneously at several auction servers developed and managed by independent organizations.

Our goal in this paper is to explore the role of accountable design as a practical complement to these classical approaches. For our purposes, an *accountable* system associates states and actions with identities, and provides primitives for actors to validate the states and actions of their peers, such that cheating or misbehavior become detectable, provable, and undeniable by the perpetrator. Note that this definition generalizes some common uses of the term (e.g., in [14]): the goal is to assign responsibility for states and actions, and not just to track and audit access control decisions. Accountability is a means to protect the integrity of the system when security mechanisms are not sufficient to prevent all forms of malicious or subversive behavior.

3. Accountability as a Means to an End

Accountability has long been viewed as a property of trustworthy computer systems [9] or as a secondary attribute of dependability [4]. We advocate elevating accountability to a *first-class* design principle for dependable networked systems. The goal of accountable design is to detect inconsistent or incorrect states or actions in a faulty component. A fully accountable system could validate a component's state as a function of the actions applied to it, enabling audits to identify the original source of a fault even if its effects spread to other components.

The basic premises of accountability are that every action is bound to the identity of its actor and that actions have well-defined semantics that enable a component's peers (such as an auditor) to reason about its states and behavior. Then a fully accountable system can ensure that the actions and state of each actor are:

- **Undeniable**: actions are binding and non-repudiable;
- **Tamper-evident**: any attempt to corrupt state is detectable;
- **Certifiable**: the correctness of states and actions can be verified.

These three properties enable participants to validate the integrity of component actions and internal state, and assign responsibility when the observed behavior does not comply with a specification. Correctly functioning components in an accountable system can provide evidence of their integrity, while dishonest or compromised components cannot hide their misbehavior.

Given these properties, accountable design becomes a *means* to meet classical dependability objectives, rather than an end in itself. Accountability properties satisfy three

of the four major means to system dependability [15]: they can be used to prevent, tolerate, and remove trust faults.

Fault Prevention. An accountable system discourages lying, cheating, and stealing by exposing an attacker to the risk of legal sanction or other damage to its interests. Effective accountability increases the cost of malicious behavior, so that the expected benefit of misbehavior is zero or negative. From a game theory perspective, the dominant actor strategy is to comply with the standards of behavior established by the system designers.

Fault Tolerance. Accountable systems offer a means to detect and isolate faults, preventing them from spreading to the rest of the system. While it may not be possible to mask a fault entirely, incorrect executions can be suppressed so that an attempt to lie, cheat, or steal is not successful. Accountable design might also allow recovery by rolling back states once a fault is detected.

Fault Removal. By identifying the root source of a fault, accountable systems can exclude misbehaving actors and limit their impact on the rest of the system.

3.1 Building Accountable Systems

The key challenge is to develop general and practical methodologies and techniques that can approach the ideal of full accountability for networked systems. The remainder of this section gives an overview of some directions we are pursuing, illustrated with examples of research systems that incorporate accountability mechanisms. Section 4 addresses some of the issues and obstacles for a general approach to accountability.

Secure hashes and digital signatures are fundamental building blocks for accountable systems. Suppose that each actor possesses at least one asymmetric key pair bound to a principal that is responsible for its actions. Public keys are distributed securely to other participants using a Public Key Infrastructure or decentralized protocols to exchange keys, manage trust, and form security associations. When actors communicate, each request or response is viewed as a binding and non-repudiable *action*. Actors sign their actions with their private keys. Participants may retain signed action records they have received, and may present them to other parties to justify their own states and actions.

Of course, signed actions are not a guarantee of intent by a principal: most security breaches occur when components are subverted or private keys are otherwise stolen [2]. In particular, an attacker that succeeds in subverting some component may attempt to tamper with its internal state and/or take improper actions. A primary goal of accountable design is to limit the damage from such attacks, by preserving a certifiable record of how a component's current state resulted from the signed actions that it received from its peers. Since the received actions are unforgeable, a subverted component cannot certify a tampered state. In this way, accountable systems can make subversion of a component apparent to its peers.

Emerging standards for interoperable service-oriented computing such as WS-Security and XML Digital Signatures now provide for digitally signed messaging based on platform-independent SOAP/XML protocols. IPsec also provides for signed IP datagrams. We emphasize that signed communication is a step beyond authenticated Transport Layer Security (e.g., SSL), which is common for secure services today. Digital signatures provide strong cryptographic evidence of a state or action that is verifiable by a third party; false accusations are computationally infeasible. US law now affirms that digital signatures are legally binding, opening the possibility of legal sanction for misbehavior. In contrast, actions on TLS-based secure channels are repudiable and cannot be proven to a third party.

Digitally signed communication alone is not sufficient for accountability. Full accountability requires that each participant maintains a cache of digitally signed records of previously executed actions, integrated with the component's internal state. Thus accountability must be "designed in" to systems and protocols. We envision auditing interfaces that allow an auditor to challenge a component to provide evidence of the correctness of its operations and state. This evidence may include sets of retrieved action records that provide cryptographic justification for state transitions and actions, according to the defined component semantics and contracts with other actors.

3.2 Separation of State and Logic

Any general approach to accountability requires a decoupling of service state from service logic, and an explicit specification of component semantics. The semantics define the effect of incoming actions on state, and which actions are legal in a given state. A peer or auditor must have sufficient knowledge of the semantics to verify that a component behaves consistently and correctly.

One direction we are pursuing is to exploit this separation to factor out common accountability functions from the application, and incorporate them into a reusable state storage layer for certified, authenticated, tamper-evident state—CATS. We are experimenting with a prototype CATS store based on a persistent authenticated dictionary [1, 17], which integrates signed action records and cryptographic state digests into a tamper-evident authenticated data structure [18].

The CATS storage abstraction enables a component or network service to certify its operations. We consider the state of a service as a collection of named variables. For the purposes of accountability, we can decompose service ac-

tions into transformations on variables, together with common operations to access and update variables. A CATS store associates service state with the signed actions that triggered recent state transitions. It certifies those transitions relative to signed state digests, which are published periodically. CATS incorporates new primitives for secure auditing: the store can answer challenges about its behavior by providing a sequence of non-repudiable action records to justify its state. The degree of auditing is a tradeoff between overhead and the probability of detection, as discussed in Section 4.

Publishing a state digest is an act of *explicit commitment*. Once a state is committed and the digest is published, it cannot be withdrawn. Any attempt to modify the state once an action is committed leaves a tamper-evident fingerprint. It is computationally infeasible to tamper with committed state in a way that is undetectable in an audit.

While CATS is a useful building block for accountable network systems, the state store alone is not sufficient to assure that a state or action is a valid result of previous inputs to the component. In effect, a CATS-based component must express its operation semantics as a state machine; CATS maintains the history to verify that component behavior conforms to the state transition rules.

The state-based approach to accountability is widely applicable to state-driven services and components that execute actions on behalf of other actors. Crash-only software [6] takes a similar approach in the context of building systems with well-defined crash behavior.

3.3. Examples

Several recent systems embody notions of accountability to build more dependable systems that detect and punish misbehavior, i.e., lying, cheating, and stealing. We briefly outline a few examples.

Samsara [8] strives to prevent cheating in peer-to-peer storage using probabilistic detection by auditing, and penalizes violators by discarding their data.

Robust Congestion Signaling [10] prevents bandwidth stealing by using nonces to detect probabilistically a data receiver's attempt to misrepresent congestion signals received from the network. The sanction for cheating is to reduce the sending rate, removing the receiver's incentive to cheat.

SUNDR [16] is a tamper-evident file system for untrusted storage servers. SUNDR clients can detect most inconsistencies and avoid using invalid data.

SHARP [11] is an architecture for secure federated resource management. It defines mechanisms to delegate control over resources. Delegation in SHARP is accountable: if an actor attempts to delegate resources it does not possess, the misbehavior can be caught and proven using undeniable cryptographic evidence.

These examples illustrate that integrating accountability techniques in system design can offer protection that goes beyond the basic safeguards of perimeter defenses. Of these examples, however, only SHARP can prove misbehavior to a third party. Accountable design builds on the techniques used in these systems and extends them with the means to assign responsibility for incorrect state and actions. Accountability lays the foundations for legal sanctions and discourages malicious behavior. In effect, accountability can serve as a form of *defense in depth* to supplement perimeter defenses, since even trusted components can be compromised and security perimeters can be breached [21].

4. Accountability Challenges

This section highlights some questions and issues pertaining to the role of accountability in dependable systems. These questions address the generality and costs of the techniques, the need to specify and validate component behavior, probabilistic accountability and the metrics to quantify it, the role of sanctions for cheating, and privacy considerations.

How general are the techniques for accountable systems?

Any approach to accountability will impact the protocols and state representations for network services. General application-independent approaches are elusive. The major constraint to a general solution is the need to define component semantics to allow explicit verification; it must be possible to detect when a component's behavior violates its semantics, and tractable to prove that any audited action is a correct response to actions the component received, given its certified state. While this approach is feasible for particular classes of services and actions, applying these techniques to complex services will require significant effort and experimentation. A state-based solution can simplify the construction of accountable systems, as described in Section 3.2.

What are the metrics for accountability?

Full accountability may be too costly and cumbersome to be practical. Since accountability focuses on detection and sanction, rather than prevention, a weaker probabilistic approach is possible and may be appropriate. Lampson [14] makes a similar argument in the context of practical security.

In a probabilistic setting we can measure accountability by the probability that malicious behavior will be detected. Verifying actions probabilistically makes accountability less intrusive and costly. In a system with strong identities

and repeated interactions, even probabilistic misbehavior detection can provide strong disincentives for cheating; actors may occasionally cheat, however, repeated cheating carries a high probability of detection, and it is not a winning strategy if sanctions are sufficiently costly to the perpetrator. Furthermore, by controlling the frequency and depth of auditing, systems can impose a bound on its overhead and provide different levels of accountability targeted to their own needs. Finally, the use of a quantifiable metric will make available a range of analytical tools that can help study accountability in a rigorous and formal framework.

Who to blame for an inconsistency?

Component integration affects accountability and may have impact on the ability to identify the perpetrator of an invalid action. When an action triggers actions on other components, its correctness may depend on the correctness of all triggered actions. To preserve the properties of accountability and be able to assign responsibility for incorrect actions, it is important that each integrated component act with its own key and be audited with respect to the actions of its clients; each action must be signed and auditable.

Note that servers with secure channels based on SSL/TLS are not accountable. In particular, a service cannot prove that an action it takes was taken on behalf of a client, unless the client has the means to sign for the actions that it originates; the server should not be trusted to sign for it. Full end-to-end accountability requires that the means to sign requests extend all the way to the originating client (e.g., a Web browser). One common but unsatisfactory approach is for the client to run trusted proxy to issue certifications for an action using the client's key (e.g., MyProxy).

Two identities share responsibility for correct execution of a component: the creator and the operator. The operator can modify the software or change the environment of the software, e.g., by reconfiguring it. To prove that the operator is responsible for any misbehavior, it is necessary to certify that the software is unmodified. In practice, it is not clear how to separate the actions of the software from the actions of the operator without using a trusted execution platform. Moreover, even if misbehavior is isolated to a component, it may be the fault of an attacker rather than the principal bound to that component.

What sanction mechanisms are necessary?

Once misbehavior is detected, the effectiveness of accountability rests on the power and severity of punishment. From a game-theoretic perspective, the major requirement for a sanction scheme is that it should make the expected cost of lying, cheating, and stealing negative. This condition is sufficient to discourage self-interested rational actors from lying, cheating, and stealing.

An approach based on reputation may be sufficient in many scenarios. Such approach seems particularly useful, when there is competition among components. Importantly, the reputation service should be resilient to lying as malicious actors might try to abuse it. Recent robust reputation algorithms such as EigenTrust [13] can tolerate large scale malicious behavior and satisfy this requirement.

A legal sanction framework might provide an alternative solution, e.g., fines for proven misbehavior. Legal solutions, however, require undeniable and strong evidence. The fact that accountable design produces undeniable, non-repudiable evidence, together with the legal acceptance of digital signatures in countries around the world, strengthens further the case and feasibility of the legal approach.

What is the cost of accountability?

The benefits of accountability come at the expense of increased computation and storage requirements. Accountable systems perform additional work to protect against malicious behavior, e.g., cryptographic operations, recording and storing actions, explicit auditing and verification of integrity and policy compliance, etc. The cost of these operations can be decomposed into three dimensions: storage, authentication, and validation overhead. The granularity of recording state information impacts the storage overhead, the choice of key length, hash function, and amount of recorded information affects authentication. Finally, the desired level of assurance is the primary component of the verification overhead. The different dimensions have different impact on the cost and guarantees of accountability and we need to study and understand this relationship better.

Accountability operations may or may not be on the critical path of execution. Synchronous verification of an operation as soon as it completes can provide maximal protection, at the expense of somewhat degraded system performance. A more relaxed asynchronous model can reduce the cost of accountability at the expense of the timeliness of detecting misbehavior. While the correct model depends on application requirements, we believe asynchronous accountability will satisfy most requirements—detection of misbehavior may be delayed, however, it is never avoided.

Privacy and anonymity?

The accountability mechanisms outlined here are directed at maintaining system integrity: they are not sufficient to hold components accountable for preserving confidential information. Encryption may limit the usefulness of disclosed information but also constrains the operations that a component can perform.

Accountability rests on the premise that there exists a binding between sanctionable identities and actors, and between actors and their actions. As such, accountability relies on the availability of an infrastructure for identity management that is secure enough to resist multiple-identity or "Sybil" attacks. While it may be possible to retain some form of anonymity, e.g. using anonymizing proxies or borrowing techniques from the digital cash domain, the need for strong identities clashes with any desire for privacy and anonymity of transactions.

Advances in authorization and privacy-preserving computation may provide solutions to these important limitations.

5. Conclusion

We view accountable design as a first-class technique dependable distributed systems. Accountability is an important system property and a means to building dependable networked systems whose components span trust domains. Integrating explicit accountability interfaces within services and components can enable participants to verify the correctness of component behavior and assign responsibility when things go wrong. Full accountability makes it possible to tolerate, detect, isolate, discourage, and remove misbehaving components. Accountability may be probabilistic: metrics for accountability offer new ways to reason about and quantify dependability of networked systems.

References

[1] A. Anagnostopoulos, M. T. Goodrich, and R. Tamassia. Persistent Authenticated Dictionaries and Their Applications. In *4th International Conference on Information Security, ISC*, October 2001.

[2] R. J. Anderson. Why Cryptosystems Fail. *Communications of the Association for Computing Machinery*, 37(11), Nov. 1994.

[3] Avizienis. The N-Version Approach to Fault-tolerant Software. *Transactions on Software Engineering*, SE-22:1491–1501, 1985.

[4] A. Avizienis, J.-C. Laprie, and B. Randel. Fundamental Concepts of Dependability. Technical Report 010028, UCLA, 2001.

[5] C. Cachin. Distributing Trust on the Internet. In *the International Conference on Dependable Systems and Networks*, pages 183–192, 2001.

[6] G. Candea and A. Fox. Crash-Only Software. In *9th Workshop on Hot Topics in Operating Systems*, 2003.

[7] M. Castro and B. Liskov. Practical Byzantine Fault Tolerance. In *Proceedings of Symposium on Operating Systems Design and Implementation*, 1999.

[8] L. P. Cox and B. D. Noble. Samsara: Honor Among Thieves in Peer-to-Peer Storage. In *the 19th ACM Symposium on Operating Systems Principles*, 2003.

[9] Department of Defense. Trusted Computer System Evaluation Criteria. Technical Report 5200.28-STD, Department of Defense, 1985.

[10] D. Ely, N. Spring, D. Wetherall, S. Savage, and T. Anderson. Robust Congestion Signalling. In *the International Conference on Network Protocols*, 2001.

[11] Y. Fu, J. Chase, B. Chun, S. Schwab, and A. Vahdat. SHARP: An Architecture for Secure Resource Peering. In *Proceedings of the 19th ACM Symposium on Operating System Principles*, pages 133–148, October 2003.

[12] H. H. Goldstine and A. Goldstine. The Electronic Numerical Integrator and Computer (ENIAC). *Annals of the History of Computing*, 18:10–16, 1996.

[13] S. D. Kamvar, M. T. Schlosser, and H. Garcia-Molina. The EigenTrust Algorithm for Reputation Management in P2P Networks. In *the 12th International World Wide Web Conference*, 2003.

[14] B. W. Lampson. Computer Security in the Real World. In *Proceedings of the Annual Computer Security Applications Conference*, 2000.

[15] J.-C. Laprie, editor. *Dependability: Basic Concepts and Terminology*. Springer-Verlag, 1992.

[16] J. Li, M. Krohn, D. Mazières, and D. Shasha. Secure Untrusted Data Repository (SUNDR). In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, pages 121–136, 2004.

[17] P. Maniatis and M. Baker. Enabling the Archival Storage of Signed Documents. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST 2002)*, pages 31–45, January 2002.

[18] R. C. Merkle. Protocols for Public Key Cryptosystems. In *Proceedings of the 1980 Sysmposium on Security and Privacy*, pages 122–133, April 1980.

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

[20] S. W. Smith, E. R. Palmer, and S. Weingart. Using a High-Performance, Programmable Secure Coprocessor. In *Financial Cryptography*, pages 73–89, 1998.

[21] J. D. Strunk, G. R. Goodson, M. L. Scheinholtz, C. A. N. Soules, and G. R. Ganger. Self-Securing Storage: Protecting Data in Compromised Systems. In *4th Symposium on Operating System Design and Implementation (OSDI 2000)*, pages 165–180, October 23-25 2000.

[22] I. E. Sutherland. Sketchpad: A Man-Machine Graphical Communication System. In *the Spring Joint Computer Conference*, 1963.

[23] A. R. Yumerefendi and J. S. Chase. Trust but Verify: Accountability for Network Services. In *Proceedings of the Eleventh ACM SIGOPS European Workshop*, September 2004.