Technical Report

UCAM-CL-TR-753
ISSN 1476-2986

Number 753

UNIVERSITY OF CAMBRIDGE

Computer Laboratory

Carbon: trusted auditing for P2P distributed virtual environments

John L. Miller, Jon Crowcroft

August 2009

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

© 2009 John L. Miller, Jon Crowcroft

Technical reports published by the University of Cambridge
Computer Laboratory are freely available via the Internet:

http://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

Carbon: trusted auditing for P2P distributed virtual environments

John L. Miller[1,2] and Jon Crowcroft[2]
[1]*Microsoft Research, Cambridge, United Kingdom*
E-mail: johnmil@microsoft.com
[2]*Computer Laboratory, University of Cambridge, United Kingdom*
E-mail: jc22@cl.cam.ac.uk

Abstract

Many Peer-to-Peer Distributed Virtual Environments (P2P DVE's) have been proposed, but none are widely deployed. One significant barrier to deployment is lack of security. This paper presents Carbon, a trusted auditing system for P2P DVE's which provides reasonable security with low per-client overhead. DVE's using Carbon perform offline auditing to evaluate DVE client correctness. Carbon audits can be used to catch DVE clients which break DVE rules – cheaters – so the DVE can punish them. We analyze the impact of applying Carbon to a peer-to-peer game with attributes similar to World of Warcraft. We show that 99.9% of cheaters – of a certain profile – can be caught with guided auditing and 2.3% bandwidth overhead, or 100% of cheaters can be caught with exhaustive auditing and 27% bandwidth overhead. The surprisingly low overhead for exhaustive auditing is the result of the small payload in most DVE packet updates, compared to the larger aggregate payloads in audit messages. Finally, we compare Carbon to PeerReview, and show that for DVE scenarios Carbon consumes significantly less resources – in typical cases by an order of magnitude – while sacrificing little protection.

1. Introduction

Distributed Virtual Environments (DVE's) are interactive, multi-node simulations. Most deployed DVE's are networked computer games such as Quake III, Halo, and World of Warcraft (WoW). These – and indeed most other deployed DVE's – are implemented as client-server applications.

Peer-to-peer (P2P) DVE architectures can improve DVE scalability, but are difficult to secure. Existing security technologies – such as Practical Byzantine Fault Tolerance[1] provide good preventative distributed system security, but at a premium. Unfortunately, BFT-based systems have significant bandwidth and latency overhead, reducing their utility in DVE's.

Auditing is a lighter weight approach which retains good – although deferred – correctness guarantees and has significantly lower impact. Auditing focuses on validating prior DVE transactions to identify errors in state communication or determination. In game terminology, auditing catches cheaters. Caught cheaters are typically punished, ideally deterring future cheaters from misbehaving.

We propose Carbon, an auditing system specifically designed to catch certain categories of cheaters in P2P DVE's. Owing to its specialization for DVE's, Carbon is lighter weight than other auditing solutions such as PeerReview[2], while retaining most relevant benefits of those systems. A key part of Carbon's solution is a trusted – rather than untrusted – auditor. We believe this is an acceptable compromise between scalability and security, and not onerous to DVE creators already used to maintaining centralized infrastructure.

The remainder of this paper describes the Carbon auditing system, and illustrates its overhead through a detailed example. Section 2 provides an overview of DVE research relevant to security, such as DVE threat modeling, frameworks with integrated security, and stand-alone security models. Section 3 outlines the Carbon auditing system, describing its main components and interfaces. Section 4 evaluates Carbon's attributes and

performance, and compares Carbon overhead with the most prominent P2P auditing system today, PeerReview. Finally, section 5 presents conclusions and future work.

2. Related Work

Distributed Virtual Environments have been researched for decades, with a significant body of research into P2P DVE's and DVE security emerging in the past eight years. Research related to security, correctness, and cheating are the most relevant to Carbon. In some cases this work is presented as an integral part of a full P2P DVE framework. In others, the work stands on its own, and can be applied to a variety of DVE architectures.

The remainder of this section presents prior art in DVE threat modeling, P2P DVE frameworks, and stand-alone DVE security research. Our analysis in section 4.3 makes certain assumptions about DVE behavior and traffic patterns, and so we briefly discuss DVE behavior in section 2.4 for the benefit of those unfamiliar with the domain.

2.1 DVE Threat Models

Several DVE cheat taxonomies and threat models have been proposed over the last ten years. Each provides insight into the types of threats faced by deployed DVE's.

Yan and Randell [3] provide a detailed network game cheat taxonomy. Their taxonomy classifies most of the cheats encountered in network game DVE's, and includes several practical examples. Most of the threats described in this section are derived from Yan and Randell's taxonomy, though they are represented in other taxonomies as well.

Webb and Soh [4] present an interesting overview of cheating and their own taxonomy. They divide attacks into four categories, based upon their level in the application stack: Game level, application level, protocol level, and infrastructure level. Examples are provided of implemented cheats in each of these categories, as well as brief discussion of applying significant mitigations from recent research to combat those cheats. Most of the salient attacks are similar to those presented in Yan and Randell's taxonomy.

Yee et. al. [5] propose a massively multiplayer online role playing game (MMORPG – a particular type of game DVE) threat model, and briefly describe existing mitigations. Their treatment – while interesting – is largely focused on human factors rather than those directly addressable by software mitigations. This makes their model useful for examining the overall milieu of publishing a game, but less so for evaluating software security.

Carbon is an auditing system intended to provide DVE's a way to identify cheaters (nodes violating DVE software rules). We've extracted the following six categories from the cheat taxonomies above. While only a subset of overall cheats, we believe these represent a significant category of threats, and mitigating them is desirable.

1. *Client Misbehavior*. Rather than detecting client modification as such, we are only concerned with externally observable behavior. In particular, that DVE state is faithfully transferred between nodes, and that communicated DVE state changes obey DVE rules.
2. *External event modification*. Some attacks modify events during their transfer between nodes. For example, one aimbot (a software agent that aims shots more quickly and accurately than a human) is implemented as a network proxy which modifies client packets to unfairly improve player performance.
3. *Collusion*. Collusion attacks are used to defeat defenses relying upon specific client behavior (such as indifference) for security. A common form of collusion attack is identity or spatial manipulation to assume a privileged role relative to another node, for example to be a member of the quorum deciding that node's state changes.
4. *Timing*. Forging timestamps on events, making them appear to have happened earlier or later than they actually did. This can provide significant unfair advantage to the cheating player, as they can make their decisions based on data they're not supposed to have, or even reverse previous decisions.
5. *Sybil attacks*. Attacks where a node sends messages masquerading as another node, for a variety of purposes.
6. *Inconsistency*. Attacks where inconsistent state values are transmitted for a given state variable, for example providing different neighbors inconsistent positions for the same object.

These attacks cover a broad spectrum of known game cheats and vulnerabilities in proposed P2P architectures.

2.2 P2P DVE Frameworks

We are unaware of P2P DVE's and P2P DVE frameworks in broad use, but several are proposed in literature. Some frameworks include security measures to help prevent cheating. Two of the most notable are SimMud and FreeMMG.

SimMud[6] proposes a DVE framework based upon traditional P2P infrastructure. It uses the Pastry[7] DHT to store and retrieve key-value pairs and organize other overlays. The Pastry-based Scribe protocol[8] provides application-layer multicast, enabling pub-sub event distribution. In terms of security, this architecture relies upon the indifference of randomly chosen parties for correct operation. State is described as key-value pairs. Authoritative state for each variable is stored at a master chosen by Pastry ID similarity to the variable key. State updates are disseminated to subscribers through Scribe trees, again constructed based on Pastry ID. A well-placed attacker can authoritatively modify state which does not belong to them, and prevent state from correctly propagating to subscribers. We believe this level of security to be insufficient in the face of motivated attackers.

FreeMMG[9] is a DVE framework built upon a combination of servers, peer specialization, and replication. Servers oversee division of the simulated world into segments, and monitor membership within each segment. Segments are collections of mutually interacting nodes, and are responsible for determining simulation state. Objects can only belong to a single segment, and all interactions within a segment are calculated by the member nodes of that segment. The primary cheat vector addressed by FreeMMG is collaboration among segment participants to subvert DVE rules. FreeMMG mitigates against this cheat by requiring a certain number of participants (e.g. k) be present in each segment. However, this can be easily subverted by a cheater with sufficient resources, either with control over k nodes, or in collaboration with $(k-1)$ other cheaters. While a practical approach, this security model relies upon the presence of at least one honest node per segment, reducing its efficacy.

Neither of these frameworks fully defend against any of the attacks listed in section 2.1. Both are significantly more vulnerable to collusion and inconsistency attacks than client-server architectures. Collusion allows an attacker to authoritatively control state, by locating the quorum or node responsible for processing state updates, and subverting it. Inconsistency attacks can be mounted by any node responsible for storing state. For example, a pastry node in SimMud could return random values for each key request made by an external party.

2.3 DVE Security Work

There are three main approaches for stand-alone protection of DVE's from cheating: protecting DVE software and communications, distributing state ownership to disinterested third parties, and auditing schemes. Kabus et. al. [10] provide a good – though slightly out of date – overview of all three.

Protecting software integrity and communications is a trusted computing base (TCB) approach. Mobile Guards[11] are a recent example of this approach. DVE software is modified to integrate a trusted software component – a *mobile guard*. Portions of the DVE are encrypted with keys only available from a functional mobile guard. Likewise, communications and data access can have integrity and confidentiality guaranteed with keys either contained within or derived from the mobile guard. As long as the mobile guard is not compromised, the system can guarantee its communications byte stream is unaltered, and DVE rules are enforced as coded both locally and remotely. Attacker compromise of mobile guards is mitigated by issuing updates more frequently than the guards can be compromised. Assuming the mobile guard is not compromised, this approach provides strong security guarantees addressing all six of the attacks described in section 2.1. Unfortunately, Mobile Guard efficacy is predicated on identifying an interval in which the mobile guard cannot be compromised, which must be sufficient for new mobile guards to be distributed. This core requirement – that you know when the TCB is compromised – is one of the issues dividing those who accept TCB's as sufficient and those who do not.

Distributing state ownership to disinterested third parties is another security technique. As mentioned in Section 2.2 above, P2P DVE's such as SimMud and FreeMMG use this method. It has been proposed separately as a mechanism for protecting DVE's. A variant presented in IRS[12] allows state to be owned by the concerned party, but verifies state update calculations by performing them at multiple untrusted nodes, then comparing the results. It assumes a disinterested third party has no motivation to break DVE rules in terms of state representation or updates, and that several disinterested parties are even less likely to collude for this purpose. Unfortunately this is not necessarily the case: third parties can maliciously tamper with data, whether it is relevant to them or not. For example, they can exploit their position to broker access to the state, requesting a fee from the data owner to keep the data secure. Or, the party with the greatest interest in a given piece of state

can manipulate the system to ensure that control and auditing of that data falls to itself. The same argument holds true for compromising quorums of disinterested third parties, though of course compromising a quorum is usually more work than compromising a single node.

If detecting – as opposed to preventing – illegal state changes is an acceptable level of mitigation, then auditing schemes can provide good DVE security. The best example of this sort of protection in recent literature is PeerReview[2]. PeerReview is an auditing system with good scalability and correctness guarantees. State changes and local transactions relevant to state calculations are stored in certified append-only local logs. Log contents are committed by peer exchange of signed log digests. Audits are performed by auditors called *witnesses*, who simulate forward from a state through a series of logged events to ensure correctness. Based on audit results and behavior, nodes are labeled as trusted (correct), suspected, or exposed (incorrect). Audit frequency guidance is provided in terms of number of witnesses and the probability of illegal activity. PeerReview offers protection against externally observable client misbehavior, external event modification, collusion below a certain threshold, Sybil attacks – since it uses RSA-key based identities – and inconsistency.

Since PeerReview is the system most similar to Carbon, we provide a comparison of salient attributes of both systems in sections 4.2 through 4.4. Our system provides similar security guarantees – with the notable exception of message non-repudiation – but has significantly lower resource requirements in the DVE scenario.

2.4 General DVE Characteristics

DVE operation is largely characterized by four activities:

1. Simulation. Evaluating state change and events. This is typically a lightweight activity, for example accepting an input and issuing an update equation.
2. Rendering. Rendering the DVE perspective to present to the participant. This is usually the most significant activity in terms of memory, I/O, and processor consumption, by an order of magnitude or more.
3. Communication. DVE nodes exchange messages to determine state changes and to refresh shadow (non-authoritative) state copies. In a P2P DVE, this traffic is typically the traffic required to describe a state change multiplied by the number of outstanding shadow copies.
4. Persistence. Storing DVE data, for example saving avatar state for later retrieval. Persistence typically involves small amounts of data, infrequently written, for example a few kilobytes once per session.

Auditing shouldn't directly affect a client node's simulation or rendering: these activities are identical at a client node with and without auditing, though additional simulation workload is introduced at auditors.

The remaining activities, communication and persistence, can be characterized by the attributes listed in Table 1.

Table 1 – Sample DVE attributes

Function	Variable
State snapshot in kilobytes	$\lvert S_i \rvert$
Average events per second	E_i
Average outgoing event bandwidth (per neighbor)	$\bar{R}(E_i)$
Average outgoing payload bandwidth (per neighbor)	$\bar{F}(E_i)$
Average shadow state copies	Q

Let $\lvert S_i \rvert$ be size in kilobytes for a node's dynamic state description, such as avatar position and attributes, and shadow state for objects being tracked by that node. Let E_i be the number of locally initiated events per second. Let $\bar{R}(E_i)$ be the average traffic in kbps required to describe a node's state transitions to a single neighbor. Since packets tend to be small and frequent, let $\bar{F}(E_i)$ be the payload portion of $\bar{R}(E_i)$, excluding packet framing overhead. Let Q be the average number of neighbors receiving shadow state updates for a given piece of state.

Incoming and outgoing non-audit bandwidth for a P2P DVE will each be at least $\bar{R}(E_i) \times Q$, because of shadow state updates. Suppose all clients initiate one event, resulting in a locally authoritative state change.

Each client transmits Q copies of its state change, one to each shadow state subscriber. Since each state variable has Q shadow copies, this implies each client is subscribed to Q times as many state variables as those it owns, and so it will also receive Q updates.

Persistence load varies depending upon DVE architecture. In some cases dynamic state is regularly saved. In others, it is stored only between client sessions. In still others it is never persisted. The most relevant attribute for DVE persistence is the state snapshot, and so the dominant factor to examine is frequency of state snapshot persistence.

2.4.1 Network Game DVE Traffic Patterns

Network Game DVE's are consumer-grade DVE's intended for home use, and optimized to provide an immersive experience with limited resources.

These immersive experiences require interactive response to inputs. Many of these DVE's are fast-paced combat simulations, with real-time activity such as aiming and firing weapons, chasing opponents, and competing for resources. These applications are extremely latency sensitive, with latencies greater than 200 ms significantly degrading the experience[13][14], and latencies of 100ms or less preferred.

These DVE's need to be able to run on most personal computers, and over most network links, which typically means functioning over dial-up connections (56 kbps). For example, both Quake III Arena and World of Warcraft can function over dial-up.

Consumer-grade DVE's are characterized by low packet inter-arrival times (responsiveness), and very small packets (low resource requirement). [15] found Quake III has typical packet sizes of 70-90 bytes, and typical inter-arrival times between 10 and 50 ms. [16] found World of Warcaft sends frequent, small packets, typically with little or no payload. Typical IP packet size ranges between 50 and 70 bytes (our measurement), with 220 ms mean inter-arrival time (their measurement).

There is one notable exception to this behavior in popular DVE's: Second Life. This DVE is a cyberspace simulation, not a game. It emphasizes user-created content. This feature consumes significantly more bandwidth, between 10 and 1164 kbps mean download bandwidth consumption, and between 13 and 74 kbps mean upload bandwidth consumption [17].

2.4.2 Characteristic Values: a World of Warcraft-style P2P game

To help put auditing overhead in perspective, we hypothesize basic requirements of a peer-to-peer version of World of Warcraft (WoW). This model is based upon World of Warcraft (WoW) traffic models from Svoboda et. al.[16], coupled with our own measurements obtained using WireShark and WoW version 3.1.

WoW clients require on average 2.1 kbps upload and 6.9 kbps download bandwidth. The current implementation of WoW uses a command / state response model for propagating state changes. We verified this by measuring bandwidth consumed by both an actor node and an observer node, noting that each observable action taken at the actor resulted in a small transmission to the server, followed by update packets being transmitted to both the actor and observer. The update packets were identical in size, and typically larger than the command packet transmitted to the server.

Most game DVE's send frequent, very small packets, often with little or no payload. For example, in WoW 57% of download packets have an empty payload. Analyzing our own packet trace, we found that in 821 seconds of activity in a popular end-game zone, 3,881 ethernet packets were received, wi1th a total size of 284,246 bytes. Our per-packet transport overhead for Ethernet and IP framing was 54 bytes per packet, which is 209,574 bytes of overhead. Our measurement shows 74% packet framing overhead, leaving 26% actual event data (i.e. $\bar{F}(E_i) = 1.79$ kbps if $\bar{R}(E_i) = 6.9$ kbps) in communications. This tells us the numbers from Svoboda et. al. provide conservative values, and are therefore a good benchmark for our purposes.

Average inter-arrival time for state update packets is 220ms, giving us $E_i = 4.545$. The size of a full state snapshot as received from the server on initialization depends heavily upon avatar location within the DVE. We measured values between 26 kB for a quiet area to 125 kB for a busy one. We choose $|S_i|$ = 62.5 kB, roughly halfway between the two extremes. We assume Q is 10, a compromise between very busy areas with dozens of mutually interacting avatars, and very quiet areas with only enough connections to maintain P2P topology. We assume the P2P version of World of Warcraft at a minimum needs to propagate state updates to affected parties, which is the basis of our traffic analysis.

3. Carbon

Distributed Virtual Environments (DVE’s) are large collections of state, and rules for modifying that state. Carbon is an auditing system allowing DVE’s to detect illegal state changes.

DVE’s must meet certain prerequisites in order to use Carbon. Section 3.1 spells out those requirements, and introduces nomenclature for discussing how Carbon interacts with eligible DVE’s.

Carbon consists of two modules: an audit client embedded in each DVE client node, and an auditor embedded in DVE code running on one or more trusted nodes. The auditor evaluates recorded DVE state, verifying legality of the simulation run as viewed at a given participant node.

Carbon is DVE-agnostic. It doesn’t understand the intricacies of how a given DVE operates. Instead, it provides a set of basic services DVE’s use to organize auditable information. In most cases, adopting Carbon requires little modification of the DVE.

The remainder of section 3 is divided into four parts. Section 3.1 outlines DVE requirements and introduces nomenclature. Section 3.2 describes Carbon audit client requirements, and the client *Reporter* component in detail. Section 3.3 describes the Carbon *Auditor* component. Section 3.4 provides an example illustrating behavior of a DVE using Carbon. **Note**: the reader may wish to skim section 3.4 before reading further, to help motivate nomenclature and design.

3.1 Nomenclature and DVE Requirements

The diversity of DVE types has lead to an explosion of different nomenclature. This section provides our preferred nomenclature (listed in Table 2), and lists a set of requirements a DVE must meet to adopt Carbon.

Table 2 – DVE Nomenclature

Symbol	Meaning
S_i^t	State at node i at time t
E^t	Event received at time t
ID_i	DVE identity for node i
$L_i^{t1,t2}$	Audit log for node i from $t1$ to $t2$
$M_{i \to j}$	Message from node i to node j

A node is a DVE instance running on a computing resource, typically providing the view and interaction point for a single avatar. State is the collection of all local authoritative and shadow state. An event is defined by the DVE itself, but is typically anything except a node state snapshot: it may be a state change, a user command, or anything else. A message is a container for DVE or Carbon information. It can contain state snapshots, events, audit log extracts, etc.

Undecorated numeric or variable subscripts refer to a specific node. Subscripts prefixed with c refer to Carbon auditors. For example, $M_{i \to cj}$ describes a message from participant node i to Carbon auditor cj. Superscripts refer to a time or time interval. For example, $L_i^{t1,t2}$ refers to an interval of node i’s log from time $t1$ to time $t2$, inclusive.

In order for Carbon to operate, a DVE should be able to simulate forward from state snapshots using events, to compare states for similarity, and to determine whether a given event is legal to apply to a given state. DVE’s which offer recording and replay of games – such as Quake III – already meet these requirements.

Formally, a DVE is a collection of state for the N active nodes $S = \bigcup_{i=0}^{N} S_i$ and a set of rules for changing that state. The overall DVE state S is a union of individual node state S_i. Individual nodes may have overlapping DVE state. Ideally one copy of a given state variable is authoritative and the rest are shadow (non-authoritative) copies, but this is not required.

An event E is a state change or command which can result in state change for a given node’s state, i.e. $S_i^{t2+\epsilon} = ProcessEvent(S_i^{t1}, E_i^{t2})$, where $t1 < t2 <= \epsilon$. Given the state of a node at any two times in the

DVE, it should always be possible to reach the successor state by taking the predecessor state and applying a series of *ProcessEvent* operations with the appropriate events.

The DVE must be able to communicate state between nodes via messages. A node must be able to initialize itself based upon a combination of local state, and received state and event messages. Again, these requirements are already met by most DVE's.

DVE state changes must be deterministic. This doesn't rule out *choosing* state changes randomly, but a given random choice must induce a deterministic change, and both the choice and change are events which should be logged.

3.2 Carbon Audit Client: "Reporter"

The Carbon audit client is a small module implementing the DVE participant node portion of Carbon. We refer to this code module as the *reporter*, since it is responsible for taking notes on the client's behavior and interaction, and reporting these to the auditor. The reporter is implemented as a library, invoked as required by DVE client code.

The reporter provides information the auditor needs to perform audits. It is a store for client state snapshots and events both generated at and received by the DVE node. Events consist of any information material for determining state changes. This typically consists of outgoing and incoming DVE messages, but may include other information such as mouse moves and key clicks, depending upon the DVE's needs.

From the reporter's perspective, state snapshots and events are opaque data blobs, stored as simple byte arrays. Neither the reporter nor the auditor components have any knowledge of how the DVE operates, or what event and state data mean.

The reporter exposes the interfaces described in Table 3 for the exclusive use of the DVE node.

Table 3 – Reporter Functions

Function	Description
Startup	Initialize the reporter
Shutdown	Shut down the reporter
Log	Add an event or state to the log
Commit	Commit to the auditor
RequestAudit	Request an audit
ProcessMessage	Process a Carbon message
RetrieveNotice	Retrieve a Carbon notification
ReleaseNotice	Release a retrieved notification

The reporter doesn't have a thread, and does not directly transmit network messages. The DVE node calls *Startup* to initialize the reporter. It provides the local node ID and a trusted auditor ID. It calls *Shutdown* to release any transient reporter data and flush data to storage.

The DVE node submits auditable events and state to the reporter via the *Log* function. *Log* takes the log data type (event / state), DVE time, and byte array as parameters. The DVE node periodically calls *Commit* to submit its most recent state snapshot to the auditor.

Upon receiving a remote message intended for the reporter – typically from a trusted auditor – the DVE node calls *ProcessMessage* with the sender ID and message payload. The two cases where this happens today are:

1. Requesting an audit log extract
2. Supplying audit results

If the DVE node wants a remote node audited, it calls *RequestAudit* with that node's ID, the minimum DVE time range to audit, and an optional state snapshot. This function would typically be called when a DVE node is informed of state which it doubts, for example with the arrival of a new avatar.

The reporter uses *notices* to communicate relevant information to the DVE node. The DVE node calls *RetrieveNotice* to retrieve a notice whenever a reporter call indicates a notice is waiting, and *ReleaseNotice* to free it.

There are two reasons the reporter returns data to the DVE node via a notice:

1. To request message transmission, for example in response to a received message, or as the result of a call to *Commit*
2. To provide audit results to the DVE node.

3.3 Carbon Auditor: "Auditor"

The *auditor* is provided as a small library used by the DVE. It runs as a trusted DVE system component with the primary purpose of accepting state snapshots and performing DVE audits.

The auditor is an advisory component. It does not directly make decisions. It provides a framework for collecting information the DVE can use to make audit decisions, and for disseminating the results to reporters. The DVE controls when an audit should be performed, audit success evaluation, and what action to take upon a successful or failed audit.

The auditor can be embedded within an existing DVE server component. Or, a new purpose-built code base can exchange messages and perform audits on behalf of the auditor. Like the reporter, the auditor has no thread of its own.

Table 4 – Auditor Functions

Function	Description
Startup	Initialize the auditor
Shutdown	Shut down the auditor
RequestAudit	Require an audit
CompleteAudit	Provide audit results
ProcessMessage	Process a Carbon message
RetrieveNotice	Retrieve a Carbon notification
ReleaseNotice	Release a retrieved notification

The table above lists auditor library functions. Most fulfill the same function as their namesakes in Section 3.2. The only new interface is *CompleteAudit*, used by the DVE server to return audit results to the Carbon auditor, along with a list of ID's to notify.

ProcessMessage can receive three different messages, each of which raises a new notice the auditor must retrieve via a call to *RetrieveNotice*.

1. A state snapshot message.
2. An audit request message.
3. An audit result message, for example a local audit result.
4. A requested log excerpt.

Each time the auditor receives a new state snapshot message, it persists the state, and retrieves the immediate predecessor and successor state snapshots for that participant – if any. This provides the basis of a state notice the DVE server can use to determine if it should perform an audit. For example, if the magnitude of changes between subsequent state snapshots seems very unlikely, it may trigger an audit.

If an audit is required, the DVE server calls *RequestAudit*. This call is authoritative since it is made by an auditor, and results in a log excerpt request for the audited node.

When the log excerpt is received and raised as a notice, the DVE server retrieves it, and performs an audit based on the earlier state notice and the log excerpt. It notifies the auditor of the result by calling *CompleteAudit* with the final state and the audit success or failure. The auditor sends a corresponding audit result notification to the audited party, and to any other participant listed in the optional audit notification list, including potentially itself.

The example in Section 3.4 below illustrates the system.

3.4 Carbon System Operation

This section provides a detailed example of a DVE using the Carbon auditing system. We assume a P2P DVE with unique participant identities. Participants can connect and disconnect from the DVE at will, resuming their activities whenever they have time. For our example, suppose the DVE requires a state snapshot every 15 minutes.

As a reminder, the system has four types of actors: A *DVE node* is a client instance. It contains a Carbon *reporter*, responsible for Carbon client activities. The DVE *server* is a trusted DVE component. It contains a Carbon *auditor*, which coordinates auditing. This is illustrated in Figure 1, with two client nodes, Alice and Bob.

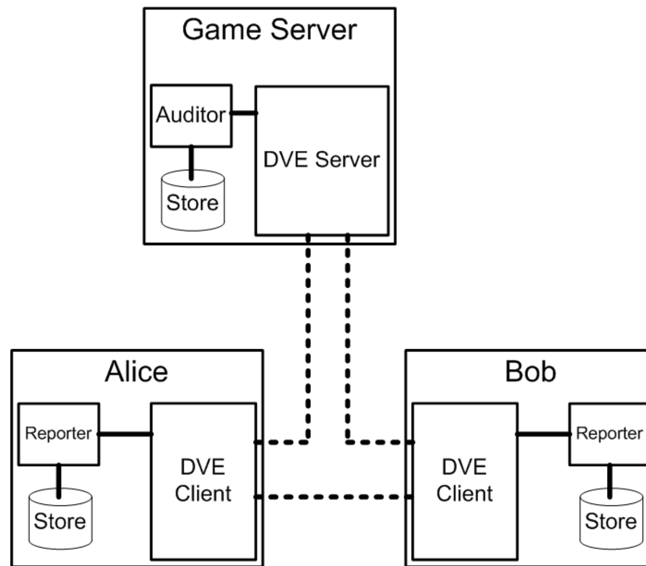

Figure 1 – Carbon System

Alice wishes to continue her avatar's virtual life. She starts up her DVE node. As part of initialization, the DVE node code calls *Startup*(ID_A , ID_{C1}), initializing the Carbon reporter with her ID and the ID of a trusted auditor.

The DVE node loads Alice's avatar and finishes integrating it into the DVE. The DVE node serializes a copy of Alice's avatar state and invokes *Log*($t0,\ S_A^{t0}$), which stores the state snapshot to the local audit log. Then it invokes *Commit()* which packages the latest state snapshot into a message for the auditor. *Commit()* signals the DVE node that a new notice is available for retrieval from the reporter. A call to *RetrieveNotice()* retrieves the message $M_{A \to C1}$ to send to the auditor. The DVE node connects to the appropriate DVE server, transmits the message, and calls *ReleaseNotice*($M_{A \to C1}$) to release its copy of the network message.

The DVE server receives the message, and ensures the sender matches the message source ID. It notes that the message target ID belongs to its hosted auditor, and invokes *ProcessMessage*($M_{A \to C1}$). The auditor deserializes the message, and saves the received state snapshot S_A^{t0} into its state snapshot table for Alice. The auditor constructs a *StateNotice* notification triple $SN\ = (S_A^{t0-k}, S_A^{t0}, \emptyset)$, and notifies the DVE server. The DVE server invokes *RetrieveNotice()* and receives the *StateNotice*. It evaluates the state snapshots, determines no audit is needed, and calls *ReleaseNotice*(SN) to return the resources back to the auditor.

Alice participates in the DVE, with her DVE node sending and receiving network messages with state changes. Her DVE node also accepts and processes her local input. Each inbound and outbound network message – with the exclusion of Carbon messages – is considered an *event*, and its payload is logged to the reporter log via a call to *Log*(t, E_A^t). The DVE can optionally record Alice's inputs for auditing. Input events can

be stored in a local event queue. Each time a network message is sent or received, the DVE empties the local event buffer contents into a new 'user input' event message, and *Log*'s it. The reporter and auditor don't differentiate between these two categories of events, though the DVE server does.

During Alice's session, her avatar encounters a new avatar Bob. When Alice's DVE node receives the message describing Bob's avatar's state $SB^{t1} \subset S_B^{t1}$, it decides to request Bob be audited. Alice's DVE node invokes *RequestAudit(*$t1$*,* ID_B*,* SB^{t1}*),* which creates a new audit request message, which her node retrieves from the reporter and sends to the auditor on the DVE server.

The auditor looks up Bob's state snapshots which fall within or immediately precede the audit interval. In this case, suppose there is a single previous snapshot $S_B^{t1-\epsilon}$. An *AuditRequestNotice* $(S_B^{t1-\epsilon}, SB^{t1}, t1-\epsilon, t1)$ is created by the auditor and retrieved by the DVE server. The DVE server compares the states and timespan, and determines an audit is warranted. The DVE server invokes *RequestAudit(*ID_B*,* $t1-\epsilon$*,* $t1$*)* specifying who and over what interval to audit. The auditor constructs a log excerpt request *(*ID_B*,* $t1-\epsilon$*,* $t1$*)* and transmits it to Bob's reporter via a notice and DVE-transmitted message, as explained above.

Bob's reporter constructs a serialized log excerpt $L_B^{t1-\epsilon,\ t1}$ of Bob's events between $t1-\epsilon$ and $t1$, then sends the auditor the excerpt as above.

The auditor extracts the message and embeds the excerpt in a *LogNotice*. The DVE server pairs this excerpt with the state snapshots it already had, and forward simulates from $t1-\epsilon$ to $t1$ checking the legality of each event as it is processed. Once the simulation time reaches $t1$, the DVE server compares SB^{t1} with its calculated version in S_B^{t1}. If the state variables specified in SB^{t1} match the value of the same state variables in S_B^{t1} then the audit passes. Otherwise it fails.

If the audit was successful, the DVE server makes a list with Alice and Bob's ID's. If the audit failed, it makes a list which includes Alice, Bob, and any other participants the DVE server wishes to notify of the audit failure, such as Bob's neighbors.

The auditor constructs a series of audit result messages containing notification of audit results, one per recipient in its list, and transmits the notifications to recipients as above.

When a reporter receives the audit result, it creates an *AuditResultNotice (*ID_B*,* $t1-\epsilon$*,* $t1$*,* $RESULT$*)* and passes it to its DVE node. The DVE node code is responsible for performing an appropriate action, such as continuing simulation, or disconnecting from Bob.

4. Analysis

P2P DVE's have more security vulnerabilities than client-server DVE's. State storage and modification is performed on untrusted peers, increasing their vulnerability to client misbehavior, collusion, and inconsistency attacks. There is no guarantee peers responsible for these activities are executing the prescribed code base and obeying DVE rules. A P2P DVE system requires means for correct nodes to ensure correctness of other nodes behavior, whether directly or indirectly.

Participants in large-scale P2P DVE's typically possess only a fragment of the overall DVE state, some authoritative, and some cached shadow state. They rely upon other DVE nodes to provide them with shadow state at appropriate times, for example when another participant moves within their area of interest.

Given a set of trusted audit servers, Carbon allows a DVE to mitigate vulnerabilities related to mis-representation of state, and to detect illegal state modification. More specifically, while the system cannot guarantee the represented state is correct, it can at least guarantee that the represented state is reachable from an earlier (trusted) state, and that the avatar presenting the state can produce an event sequence which reaches the represented state. This provides partial mitigation for client misbehavior, collusion, and inconsistency attacks.

The remainder of this section analyzes the behavior and overhead of Carbon. Since PeerReview is closely related to Carbon in terms of intent and behavior, we compare and contrast the overhead of PeerReview as well.

Section 4.1discusses Carbon efficacy in terms of the threats we outlined in section 2.1. Section 4.2 provides some details on PeerReview's behavior, and high-level comparison with Carbon. Section 4.3 provides finer grained details about Carbon and PeerReview's client resource consumption in the DVE scenario. Finally, section 4.4 discusses Carbon auditor resource consumption.

4.1 Audit Coverage

DVE's implement deterministic state machines. Given access to a state snapshot and an event, any node can determine the resulting state. This principle provides the basis of our auditing solution.

P2P DVE nodes usually perform similar activities to one another, with similar levels of trust, ideally none. In some architectures a subset of nodes are granted additional responsibilities for coordinating DVE activities, but such responsibilities are typically based upon node resources rather than trustworthiness.

Carbon provides an auditing framework for detecting invalid state transitions within the DVE. The DVE can use Carbon-provided data to perform audits, or more complex analysis, such as detecting illegal input devices.

Carbon's goal is to enable a DVE to ensure avatar integrity and correctness. It is impossible to verify avatar state integrity in isolation: the avatar's state is affected by its environment. A system examining only avatar state lacks context to verify it. For example, suppose avatars have a 'money carried' property, and Alice violates DVE rules by modifying her avatar to have a million dollars. If the auditing system evaluates only avatars, and if there is any non-avatar source of money – for example money lying on the street – Alice could claim upon audit that her million dollars was found on the street, with no way to disprove her claim. By increasing the audit scope to include Alice's entire DVE node state, the auditor has access to context which can help validate or refute Alice's avatar state: it can review her node's simulation to learn about any money on the street, and can confirm the amount is appropriate. If the provenance of the money is suspect, its source in the DVE can also be audited, and so on.

In section 2.1we described six categories of attacks a DVE security system should mitigate. These attacks are listed again below, with a brief discussion of Carbon's efficacy against them.

1. *Client modification*. Carbon cannot detect modification or replacement of remote client software: in general this is not possible without a TCB. However, Carbon does ensure externally observed client behavior respects DVE rules. These rules are typically related to faithfully executing network protocol, and state storage and transition. Carbon partially mitigates this attack. If the network protocol is violated, then clients participating in the protocol will see unexpected messages. These messages are logged at both the sender and receiver, and the auditor can detect an inconsistency when either party is audited. This in turn can trigger an audit of the compromised node, which reveals its misbehavior. Carbon also detects illegal state transitions. Audits evaluate state transitions at each node, ensuring they follow DVE rules. If they do not, then the auditor can detect this, and flag the node as violating DVE rules.
2. *External Event Modification*. Carbon cannot detect event modification within a node, such as an aimbot which inspects system memory and modifies local user input. However, it can prevent attacks which work by externally modifying events, such as a network proxy-based aimbot. If messages are modified outside the client, the client is unaware of the value of those messages, and cannot adjust its logged message copy or state accordingly. This leads to state inconsistency between nodes, and inconsistent messages, both of which are detectable by Carbon auditing.
3. *Collusion*. The collusion attacks we're concerned with involve nodes collaborating to selectively subvert or ignore DVE state transitions. If any DVE state is changed without adhering to externally observable DVE rules and both parties log their transactions, auditing can detect and expose the parties who illegally changed that state, regardless of how they are colluding.

   Specific categories of collusion require additional checks by DVE clients to detect. For example, suppose Alice transfers a million dollars to Bob, then clears that transaction from her state and audit log. If Bob retains this transaction, then individually each node's log would appear correct, but a million dollars would have been duplicated. Such attacks can be detected by comparing state snapshots of interacting clients during audits. For example, Bob's auditor could flag the transaction with Alice as significant, and from its own audit session request an audit of Alice for the same interval, providing its calculated state for Alice's avatar as the state snapshot to compare against. Alice's audit would fail, because the state provided by Bob's auditor would differ significantly from the state calculated by Alice's auditor. Note that questions of message authenticity (for example, did Bob forge the transaction from Alice, or is Alice lying?) needs to be addressed by the DVE. The PeerReview auditing system integrates this protection globally via authenticators, but at a dramatic cost as discussed in sections 4.3 and 4.4. We believe selectively protecting against this attack is a better compromise for most DVE's in terms of cost vs. benefit.
4. *Timing*. Timing attacks involve forging timestamps on packets. Carbon can defend against a subset of this attack. To circumvent Carbon auditing, an attacker would create a message, forge the

timestamp, and rewrite their local audit log to place the message in the correct location in the log. So long as the log was not committed to the auditor in the interim, this attack is undetectable. However, modifications which cross a commit boundary could be detected. It's worth noting that PeerReview provides stronger protection against this attack, so long as the attack results in a reordering of transmitted messages.

5. *Sybil attacks*. Carbon uses a unique numeric identity for each node, but relies upon the DVE to ensure those identities are sufficiently strong, and that transactions between DVE nodes are suitably identified and secured. While Carbon doesn't directly address inter-node communications and therefore Sybil attacks, meeting its identity requirements in a reasonable fashion provides a basis for the DVE to implement standard Sybil attack defenses.
6. *Inconsistency*. Carbon mitigates this attack. Nodes periodically commit their logs to the auditor. A state commit can only match a single sequence of events. If a cheating node sends inconsistent state representations to two nodes, only one of those representations can possibly match the auditable event stream the attacker commits to. Each node receiving state from the cheating node can request an audit, including the state supplied to them by the cheater. If the resulting audit doesn't match the provided state – which MUST be the case for all but one of the receivers of different state – then the audit fails and the cheater is exposed. Note that signing transmitted state snapshots may be required to make assertions of received state sufficiently strong.

Exhaustive auditing via Carbon reliably detects most of these attacks with reasonable overhead, as discussed in section 4.3 and 4.4. If less expensive auditing is desired, DVE's can leverage participant affinity for their avatar, assigning punishment sufficient to deter cheaters who believe they will be caught, and reducing the fraction of total transactions audited. The DVE should adjust punishment severity and audit probability until an appropriate deterrence is achieved. For example, a 50% chance of having their avatar's lifespan cut short may be sufficient to deter most cheating.

As mentioned earlier, Carbon does not directly evaluate DVE state correctness. Instead, it collects and organizes information for the DVE to determine when an audit should be performed. Likewise, performing the actual audit is left up to the DVE code itself. An audit can be as straightforward as verifying successive state transitions are legal, or as complex as correlating state transitions between multiple DVE views from multiple participants, or performing deep data mining to uncover more elusive DVE violations such as account sharing or theft [18] and dependency hacks such as wall hacks [19].

4.2 Comparison with PeerReview

PeerReview is a decentralized auditing system with good scalability and correctness guarantees. State changes and local transactions relevant to state calculations are stored in certified append-only local logs. Log contents are committed by peer exchange of signed log digests, and periodically uploaded to untrusted auditors called *witnesses*. *Witnesses* simulate forward from a state snapshot through a series of logged events to ensure simulation correctness. Based on audit results and general behavior, nodes are labeled as trusted, suspected, or exposed (bad).

Carbon and PeerReview have many similarities.

- Both are auditing systems, focused on error detection, not error prevention.
- Both are asynchronous, with auditing performed out of band.
- Both systems require client nodes to capture system snapshots and events, and to provide those events to auditors.
- Both use third party auditors who, given a state snapshot and a related log excerpt, forward simulate through the log excerpt to ensure behavior over the simulated interval is correct.

Carbon and PeerReview have a number of differences, as well.

- **Auditor Trust**. PeerReview can operate even when some auditors misbehave. Carbon trusts auditors to be correct.
    - PeerReview: Does not trust individual auditors, and is resilient to a limited number of auditor errors.
    - Carbon: Trusts the auditor, and requires operating auditors be correct.
- **Client Log Security**. PeerReview constructs tamper-evident logs with per-entry tamper protection. Carbon achieves tamper evidence by submitting hashes over large sets of log entries to the trusted auditor.

    - PeerReview: Allows log validation between any two events, and auditing forward from any state snapshot
    - Carbon: Allows log validation between any two submitted hashes, and auditing forward from any state snapshot.
- **Message Security**. PeerReview provides a mechanism for securing actual message exchanges between clients. It effectively creates a signature (“authenticator”) for each message, and requires a signed response. This imposes a specific structure on transactions, but provides non-repudiation, tamper detection on transmit, and many other desirable properties.
    - PeerReview. Message signing and positive acknowledgement of every packet provides message non-repudiation, and enables easy detection of certain classes of cheat such as omitting key messages from sender logs, but still reporting them on the receiver.
    - Carbon. No network message security is provided. Instead, it is expected the DVE will have its own transport security, and if desired, perform message validation as part of event auditing.
- **Completeness**. PeerReview typically audits all events by all clients in the system. Carbon uses application feedback to selectively audit suspected intervals.
    - PeerReview: Exhaustive validation of all system behavior. Probabilistic interval selection proposed but not described.
    - Carbon: Selective validation. Validation explicitly guided by the application and its clients.
- **Inter-client impact**. PeerReview affects every inter-client interaction to provide message delivery guarantees. Carbon does not, but we believe it provides sufficient security for DVE’s.
    - PeerReview: Requires a signed log commit (authenticator) for each transmitted message, and a similar receipt acknowledgement.
    - Carbon: Does not affect inter-client traffic.
- **Implementation**. PeerReview’s implementation uses sub-classing from auditor defined classes, with an audit event engine driven by PeerReview. Carbon is implemented as an API with audit event engine driven by the application.
    - PeerReview: Has total control over audit process, but significant structural and integration requirements.
    - Carbon: Application links to Carbon libraries and calls API’s as it sees fit. Carbon has no execution threads of its own.

The remainder of this section provides quantification of some of the differences between Carbon and PeerReview, as they pertain to performance. These sections rely heavily upon our description of typical DVE characteristics and our WoW-like example in section 2.4. Section 4.3 describes client resource impact of adopting Carbon, and compares it to the overhead required to adopt PeerReview. Section 4.4 provides a similar analysis for auditor resource requirements.

4.3 Carbon and PeerReview Client Overhead Analysis

Adopting an auditing or security scheme can have a significant impact on DVE resource consumption. Game DVE’s are performance-hungry, and many consumers buy high-powered computers specifically to improve their DVE performance. Any DVE considering a security model needs to pay careful attention to the overhead that model induces.

This section describes the client overhead associated with Carbon, and contrasts it to the overheads for adopting PeerReview for this scenario. It’s worth mentioning that DVE traffic patterns – small, frequently sent packets – are a worst-case scenario for PeerReview, and a best-case scenario for Carbon. The former is a side effect of the broad applicability of PeerReview to distributed network protocols, while the latter is an intentional design criteria for Carbon.

We divide our analysis of client overhead into four categories: network bandwidth, message latency, client persistence workload, and CPU overhead.

4.3.1 Bandwidth

Auditing typically involves transmission of information to auditing parties, and possibly to other clients. The most obvious impact of this activity is on client bandwidth consumption.

Carbon induces additional client network load consisting of:

- Copying the payload for non-carbon messages sent or received to the client's auditor, when requested by the auditor. If 10% of all transactions are audited, then 10% of payloads will be copied to the auditor.
- Periodic state commits, uploaded regularly to the auditor

For DVE's which exchange large amounts of state information, full auditing with Carbon increases overall client traffic to roughly double its original amount. For more typical DVE's which have small, frequent exchanges of state, Carbon overhead is partially mitigated by the more efficient framing of data sent to the auditor: Whereas small payloads are frequently transmitted for normal DVE interaction, auditing can bundle up many payloads before transmitting them to the auditor, resulting in reduced packet framing costs relative to the payload size. For our example of a WoW type P2P game, this reduces additional bandwidth consumed for propagating events from 100% additional overhead to 27% additional overhead.

A subset of PeerReview overhead is similar to that for Carbon, as it must propagate events and state snapshots to auditors. However, PeerReview typically has a quorum of auditors for each client, so each client transmits these messages several times. In addition, PeerReview has several additional sources of overhead, as described below.

- A PeerReview *authenticator* must be included in each DVE message. The authenticator is a sequence number / SHA-1 hash pair signed with a 1024-bit RSA key, with a total size of 156 bytes. This is required for each outgoing event message. The authenticator is acknowledged by the recipient with a new authenticator roughly the same size, but which needs no additional acknowledgement.
- Each received authenticator is transmitted to every auditor responsible for the authenticator creator.

Suppose both auditing systems have a state snapshot interval Z, and a probability p of auditing the interval between any two subsequent state snapshots. For subsequent analysis, let us assume $Z = 900$ seconds, and $p = 1$.

In the terms outlined earlier, Carbon bandwidth overhead can be characterized as $(|S_i|/Z) + p \times \overline{F}(E_i) \times (2Q)$. PeerReview overhead, assuming a single trusted witness per client rather than a quorum, would be $(|S_i|/Z) + (3 * E_i * 156 * 8/1024 + p \times \overline{F}(E_i)) \times 2Q$ kbps, or $(3.744 \times E_i \times 2Q)$ kbps more than Carbon. Plugging our numbers from the WoW-type game above, this gives us Carbon auditing overhead of 37.4 kbps, and PeerReview auditing overhead of 340 kbps, nearly an order of magnitude difference.

It's worth noting that auditability of an interval for both Carbon and PeerReview relies upon having an initial state snapshot, plus a contiguous series of audit log entries through the audited interval. Ideally the audit should include a final state snapshot to compare against audit results, although this isn't strictly necessary.

This has significant implications for PeerReview auditing interval selection. Even though PeerReview provides a tamper-evident log which can be examined between any two intervals, the reality is that other than verifying the sequence of messages matches the authenticators for any given sub-interval, no auditing can be performed except for an interval starting at a state snapshot. In our example above, this means auditing must be performed continuously from the most recent state snapshot preceding the interval of interest, providing no state auditing advantage over Carbon.

On the other hand, the extra overhead of PeerReview authenticators enables certain protections which are weaker or not present in Carbon.

4.3.2 Latency

Carbon introduces no latency into most client interactions, as it does not modify existing message exchange. The only place where latency may be introduced is when the packets for additional exchanges with the auditor collide with other DVE traffic. Since auditor traffic is infrequent – on the order of an exchange once every several minutes – the overall effect is negligible. Further, the Carbon model allows the application to choose when audit traffic is generated and transmitted, so audit transmissions can be delayed until a time the application finds convenient.

PeerReview adds an authenticator to every inter-client event message, resulting in negligible but non-zero delivery latency. Specifically, each packet has an extra 156 bytes of data, which on a 1 Mbps circuit requires just over a millisecond of extra transit time before the packet is fully received and can be decoded. The authenticator also requires crypto to create, which according to PeerReview's analysis, can induce another 1.5 ms of latency in circa 2007 hardware. This results in a minimum latency impact of 2.5 ms, negligible in most WAN scenarios.

4.3.3 Storage

Both Carbon and PeerReview induce client storage overhead for client logs, roughly equal to the node's network traffic, plus auditing overhead. In our example, a typical week of WoW usage (614 minutes) [20] for a Carbon DVE client would consume approximately (614 * 60) s * 37.4 kbps = 172 MB of audit client storage per client. A PeerReview audit log for the same time period would consume 1.57 GB of audit client storage, a larger but still acceptable amount. The difference in log size is because of the authenticator required for PeerReview. Each authenticator is relatively small, but with an average event payload measured in tens of bytes, two 156-byte authenticators per payload represents significant transfer and storage overhead.

4.3.4 CPU

Client CPU impact is not significant for Carbon, but may be an issue for PeerReview.

For Carbon, audit entries are retained in the local reporter log, but no cryptographic operations are performed upon them, except for a hash every snapshot interval (once per fifteen minutes in our example above).

For PeerReview, each client event message sent or received results in calculation of two non-signature hashes, an RSA signature, and an RSA signature validation. While hashing is relatively quick, signing time can quickly add up, especially for periods with high event rates.

PeerReview quotes a signing and validation time of 1.5 ms. Each client must sign and verify $E_i \times (Q + 1)$ messages per second. This consists of the E_i event messages it originates and transmits to subscribers, and the $E_i \times Q$ event messages it receives from other clients and for which it must verify the authenticator and create its own signed acknowledgement. In our example above, this is on average 50 messages per second, for a total CPU overhead of 75 ms per second. While not prohibitive on average, DVE client load has spikes which can easily be an order of magnitude larger than average, for example during a WoW battleground melee. This would raise CPU overhead to 750ms per second, or 75% of one core, enough to significantly impact DVE client performance.

4.4 Carbon and PeerReview Auditors and Overhead

The Carbon auditor is a trusted component which should be encapsulated within a trusted DVE server. If no such component (such as a rendezvous server) exists, then one may need to be built. At a minimum, the trusted DVE server needs to be able to exchange network messages with DVE clients on behalf of the Carbon auditor. It should also be able to authenticate connections and correlate the remote party with Carbon identifiers.

The PeerReview auditor (Witness) is an untrusted component. Faults are detected by statistical guarantees based upon the probability of a given auditor being faulty, e.g. cheating. The quorum size of auditors per client is determined by the probability of an auditor being faulty and the risk tolerance of the system.

For our PeerReview analysis, we assume no PeerReview auditors are faulty, and so a single auditor is sufficient for a given client. This is the best case for PeerReview overhead.

Auditor overhead can be described in terms of bandwidth, storage, and CPU. We don't consider latency, as auditors do not directly participate in DVE transactions, instead auditing out-of-band. Any latency introduced between auditors is largely irrelevant to client and overall DVE performance.

For both systems, the DVE provisions a number of auditors sufficient for its needs, ranging anywhere from a single auditor auditing all clients to one auditor per client. Each audit begins with a state snapshot, and forward simulates through a series of logged events, checking each state transition for correctness. The audit can optionally conclude by comparing resulting state to another state snapshot from the audited client.

Since the client node was able to simulate its portion of DVE state based upon the event stream captured by auditing, we know that a single auditor can replay that client's view of state and state changes. This implies each audit can typically be done without regards to any other audits, and so are fully decomposable from one another. In some cases it may be desirable to cross-check a subset of audits, which may reduce decomposability.

In addition to the checks above, PeerReview checks received authenticators to make sure messages are accurately logged by both sides of each exchange. This defense prevents attacks where one side or the other omits a message from their audit log, but not attacks where both do. Carbon does not provide this protection. Instead, Carbon relies upon the DVE to take any action it deems necessary to validate message exchange, for example auditing nodes with significant interactions on the same auditor, and comparing intermediate states to ensure no significant differences.

The remainder of this section describes the bandwidth, storage, and CPU requirements for an auditor. For our examples below, we continue evaluation based on the P2P World of Warcraft example, and further specify the DVE population as $W = 10{,}000{,}000$ total players, with a steady-state population of $X = 100{,}000$ active client nodes. For our analysis we specify $Y = 100$ auditors are active, for a ratio of auditors to active clients of 1 : 1,000.

4.4.1 Bandwidth

All Carbon client transactions are exchanges between clients and auditors, and so auditor bandwidth originating at clients is the aggregate of reporter bandwidth requirements. At a minimum this consists of commits, but may also include client audit requests, submission of requested log excerpts to auditors, and notification of audit results to clients. This traffic is detailed in section 4.3.1. Average auditor server bandwidth is (X/Y) times the average reporter client bandwidth. Using our example values above, average per-auditor bandwidth required would be 3.14 Mbps to have a 99.9% chance of detecting cheaters sometime in their avatar lifetime, or 37.4 Mbps for exhaustive auditing, both easily within the capabilities of a low end server.

PeerReview traffic analysis is more complicated in the general case, but somewhat simplified by our assumption of fully trusted, non-redundant auditors. Client message authenticators are created, written into logs, and then transmitted to a remote party. Both the sender and receiver log the authenticator, and the receiver transmits the received authenticator to the sender's auditor. Whenever an auditor performs an audit, it retrieves the log from the audited party, which contains similar data to that used by Carbon, plus the associated authenticators. Upon receiving a log excerpt, the auditor forwards any authenticators contained within the log – but not created by the log creator – to the auditor for the party who created the log excerpt. In addition, auditors periodically request contiguous sequences of authenticators from clients to verify log continuity and commitment. There is also a challenge protocol to verify nodes are live and responding to messages.

Since the continuity check and challenge protocol provide functionality not in Carbon, we ignore them for the purpose of comparative bandwidth estimation. State snapshot, log excerpt, and authenticator exchanges are the remaining traffic sources for auditors. We can calculate traffic numbers divided into two categories: client/auditor exchanges, and auditor/auditor exchanges.

Using our examples above, calculations from section 4.3.1, and assuming exhaustive auditing, the per-client contribution to each auditor is 340 kbps. An additional copy of each authenticator is exchanged between auditors, for an additional per-client overhead of $(2.496 \times E_i \times 2Q) = 227$ kbps. Total bandwidth required per auditor is 567 Mbps for our scenario, approximately 15 times as much as Carbon. As is the case for clients, the vast majority of this traffic overhead consists of authenticators.

4.4.2 Storage

Carbon auditors are responsible for storing state snapshots S_i on behalf of each client i, submitted as determined by the DVE client, but typically with a frequency Z. Based on our example DVE characteristics and steady-state population, this would result in an average of $(7 \times 24 \times 60 \times 60) \times (X/Y) \times |S_i| / Z = 42$ GB of data stored per server per week, a trivial amount of storage for modern servers. Audit log excerpts sent from reporters to satisfy audit requests are temporarily retained, and then discarded after the audit is completed, so they are not factored into this total.

PeerReview auditors typically retain a copy of logs submitted to them by clients, and must check any received authenticators against those logs. Only one copy of each authenticator needs to be retained by each auditor, but it appears twice: once in the sender's log, and once in the receiver's log. Total storage required per server per week would be $(7 \times 24 \times 60 \times 60) \times (X/Y) \times 226.5\ kbps/8 = 17.1$ Terabytes, more than 400 times as much data as Carbon. This makes retaining full client logs at each of the PeerReview auditors impractical. Indeed there is little value in retaining them after a successful audit.

4.4.3 CPU

Both Carbon and PeerReview auditors would consume CPU performing forward simulation in the course of auditing. This workload varies from DVE to DVE, but would be similar between both systems. Most DVE's consist of state changes which, given a base state and an event, are straightforward to calculate. Most CPU on DVE nodes is applied instead to rendering the DVE scene for the client, and as mentioned earlier rendering is irrelevant to auditing.

For exhaustive auditing, each audit server needs to be able to simulate client state changes at the same rate it receives new state snapshots. In other words, for each state snapshot that represents Z seconds of changes, the auditor would need to retrieve the previous state snapshot for that node, request and receive the audit log excerpt

for that period from the node. It would need to verify the audit log hash matches the committed hash for that interval, and then forward simulate through the events, checking each event for DVE legality, and comparing the submitted final state with the locally simulated final state. If the steady state ratio of clients to servers is (X / Y), then each server needs to be able to simulate at least (X / Y) DVE instances in real-time.

PeerReview auditors have an additional CPU overhead introduced by the requirement to verify authenticators. Based on the figures quoted in section 4.3.4, real-time verification of authenticators contained in the logs received from 14 nodes – at 7.5% CPU per log – would saturate a single core. This is probably the biggest single obstacle to assigning multiple clients per auditor in the PeerReview scheme. PeerReview mitigates this by not requiring auditors be trusted, hence enabling (for example) every client to also be an auditor. Unfortunately, creating a quorum of auditors per client multiplies both auditor and client resource requirements.

In general, CPU will probably be the limiting factor for audit servers for both Carbon and PeerReview. Carbon can compensate for this by reducing audit frequency from exhaustive to guided, based on the relative difference between state snapshots, and on audit requests submitted by detectors at client nodes.

5. Conclusions

Carbon is a trusted auditing system designed specifically for P2P DVE's. Running the Carbon auditor as a trusted rather than untrusted entity allows a Carbon-enabled system to distribute most operations amongst DVE participants, while retaining good error detection guarantees.

Carbon is able to detect many significant forms of cheating in DVE's, allowing it to serve as part of a P2P DVE's security system. Auditing can detect a variety of common attacks, include illegal state transitions, inconsistent representation of state to peers, and tools such as aimbot proxies which modify inter-node messages. It also obviates collusion as a way of modifying DVE state without detection.

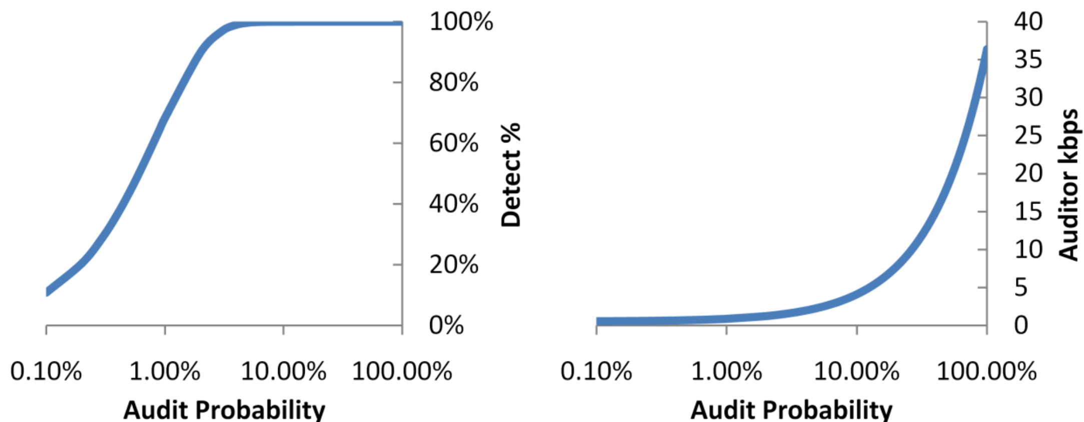

Figure 2 – Detection probability and audit bandwidth

Carbon achieves this with a modest amount of client overhead. For example, a P2P DVE which followed eventing and player patterns similar to World of Warcraft can have a 99.9% chance of catching an occasional cheater sometime in their lifetime with 7% auditing, assuming a player lifetime of 1,000 hours and cheating once per hour. This level of protection would induce a bandwidth overhead of 3.14 kbps on total non-audit traffic of 138 kbps, or 2.3%. Exhaustive auditing would require 37.4 kbps, a bandwidth overhead of 27%, still a reasonable margin. Figure 2 shows the relationship between audit probability and chance of catching a cheater, and the auditor bandwidth impact of choosing that percentage.

Client CPU overhead would be negligible, and client latency would be unaffected, as Carbon does not modify client-to-client interactions and auditing is performed out of band.

Client storage overhead is also reasonable. If we consider a P2P version of World of Warcraft as a usage model, Carbon requires 172 MB per week to store a typical players audit logs. The system could be configured with an audit log horizon (such as a week) to minimize client-side storage. Given that a game like World of Warcraft can require 5 GB of storage for basic installation, this is less than 4% overhead compared to static installation files.

We compared Carbon performance overhead and security to that provided by PeerReview. We found that in the domain of DVE's, Carbon provides similar protection with less than a tenth of the network traffic, less than a hundredth of the storage requirements, and significantly less CPU than PeerReview requires. This suggests that Carbon is a better choice for auditing DVE's, as it was designed to be.

For future work we plan to extend Carbon to provide inter-audit security to peers. In particular, we would like to enable peers encountering an adversary for the first time to be able to trust the state presented by that peer.

References

[1] Miguel Castro and Barbara Liskov, "Practical Byzantine Fault Tolerance," in *OSDI '99 Proc.*, 1999, pp. 173-186.

[2] Andreas Haeberlen, Petr Kouznetsov, and Peter Druschel, "PeerReview: practical accountability for distributed systems," in *SOSP '07 Proc.*, pp. 175-188.

[3] Jeff Yan and Brian Randell, "A systematic classification of cheating in online games," in *Netgames '05 Proc.*, 2005, pp. 1-9.

[4] Steven Daniel Webb and Sieteng Soh, "Cheating in networked computer games: a review," in *DIMEA '07 Proc.*, 2007, pp. 105-112.

[5] George Yee, Larry Korba, Ronggong Song, and Ying-Chieh Chen, "Towards Designing Secure Online Games," in *AINA '06 Proc.*, 2006, pp. 44-48.

[6] Björn Knutsson, Honghui Lu, Wei Xu, and Bryan Hopkins, "Peer-to-Peer Support for Massively Multiplayer Games," in *INFOCOM 2004 Proc.*

[7] Antony Rowstron and Peter Druschel, "Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems," *Lecture Notes in Computer Science*, vol. 2218, pp. 329--?, 2001.

[8] M. Castro, P. Druschel, A. Kermarrec, and A. Rowstron, "SCRIBE: A large-scale and decentralized application-level multicast infrastructure," *IEEE JSAC*, vol. 20, pp. 1489-1499, 2002.

[9] F. R. Cecin et al., "A Scalable and Cheat-Resistant Distribution Model for Internet Games," in *DS-RT '04 Proc.*, 2004, pp. 83-90.

[10] Patric Kabus, Wesley W. Terpstra, Mariano Cilia, and Alejandro P. Buchmann, "Addressing cheating in distributed MMOGs," in *NetGames '05: Proc.*, pp. 1-6.

[11] Christian Mönch, Gisle Grimen, and Roger Midtstraum, "Protecting online games against cheating," in *NetGames '06 Proc.*, p. 20.

[12] Josh Goodman and Clark Verbrugge, "A Peer Auditing Scheme for Cheat Elimination in MMOGs," in *NetGames '08 Proc.*

[13] Tom Beigbeder et al., "The effects of loss and latency on user performance in unreal tournament 2003," in *NetGames '04 Proc.*, pp. 144-151.

[14] Tristan Henderson and Saleem Bhatti, "Networked games: a QoS-sensitive application for QoS-insensitive users?," in *RIPQoS '03 Proc.*, pp. 141-147.

[15] Tanja Lang, Philip Branch, and Grenville Armitage, "A synthetic traffic model for Quake3," in *ACE '04 Proc.*, pp. 233-238.

[16] Philipp Svoboda, Wolfgang Karner, and Markus Rupp, "Traffic Analysis and Modeling for World of Warcraft," in *ICC '07 Proc.*, pp. 1612-1617.

[17] James Kinicki and Mark Claypool, "Traffic Analysis of Avatars in Second Life," in *NOSSDAV '08 Proc.*, 2008, pp. 69-74.

[18] Kuan-TaChen and Li-WenHong, "User Identification based on Game-Play Activity Patterns," in *NetGames '07 Proc.*, pp. 7-12.

[19] Peter Laurens, Richard F. Paige, Phillip J. Brooke, and Howard Chivers, "A Novel Approach to the Detection of Cheating in Multiplayer Online Games," in *ICECCS '07 Proc.*, pp. 97-106.

[20] Neilsen The Neilsen, "Trend Index - Video Games," no. http://en-us.nielsen.com/rankings/insights/rankings/video_games, 2009.