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

Distributed Virtual Environments (DVE's) are used for purposes ranging from military simulations to computer games. P2P DVE's enhance scalability, but are more vulnerable to attack than their client-server equivalents. We introduce Carbon, an auditing system for P2P DVE's. Carbon uses state snapshots, per-client event logs, and a DVE-controlled auditing threshold to audit DVE participant activity for legality and consistency. Carbon allows most DVE work to be distributed amongst untrusted peers, while providing centralized-quality guarantees of state correctness. Carbon overhead against a typical MMOG traffic pattern is discussed with surprisingly practical results. For P2P DVE's with state description and update characteristics similar to World of Warcraft, we show Carbon can produce exhaustive trusted audits with only 24% additional traffic per node, and sufficient selective audits with 1.3% additional traffic per node. We also show that trusted auditors can decompose DVE auditing problem, enabling linear audit scalability.

1. Introduction

Virtual Environments (VE's) are simulations typically intended to mimic a real or imagined environments. A simulated walkthrough of a building is an example of a Virtual Environment. The simulation is viewed from the perspective of an "avatar," a point of presence representing the participant. VE rules determine where and how the avatar can interact with the simulation. The VE is comprised of simulation state, and rules for updating and rendering that state. Simulation state includes the avatar's position and orientation, environmental elements such as walls, floors, lights, and the position and orientation of dynamic elements such as doors.

Some Virtual Environment simulations provide a shared interactive experience for multiple participants, each represented by an avatar. If participants are not co-located, it is usually necessary to use a Distributed Virtual Environment, or DVE. A DVE is distinguished from a VE by having state distributed among and rendered by more than one computational resource, for example a collection of networked computers.

DVE's are used for a variety of purposes, such as military simulations, immersive educational and therapeutic environments, cyberspace such as Second Life, and networked games. Networked games are by far the most common DVE. Blizzard Entertainment's World of Warcraft, for example, is a DVE with more than eleven million paying subscribers, and more than a million active avatars at its busiest times.

The distributed nature of DVE's introduces significant complications to every aspect of simulation.

Consider Quake III Arena, a game DVE which can run as either a single-player VE or a multi-player DVE. VE mode has a single game state copy, including variables such as avatar location, health, and items; the position and actions of AI avatars; and the state of mutable objects within the DVE. The avatar view is rendered from the authoritative state, and interactions are resolved directly against that authoritative state.

In multi-player DVE mode, Quake III Arena retains authoritative state at a central server, but it allows other participants to join the DVE from their own computers via the network. Networked participants interact with a local, non-authoritative state replica known as *shadow state*. Some avatar actions are speculatively executed and rendered against local shadow state, while others are resolved by forwarding a command to the authoritative server and receiving a state change in response.

DVE's can be dramatically different from a single-node VE's in terms of consistency and correctness. Shadow state lags behind authoritative state. In some cases shadow state is speculatively modified and rendered before central authority approval. In the worst

case, these changes are rejected by the central authority, and the client must roll back its shadow state to match the authoritative state. For DVE participant Alice, shadow state of DVE participant Bob lags behind Bob's perception of that state by the transit time from Bob to the server to Alice, plus any framing and processing overhead. Shadow state is almost always out of sync in minor ways, requiring complex heuristics to compensate and provide participants with similar experiences.

In the move from VE to DVE, security becomes a significant issue. In a VE, state belongs to the same entity rendering and interacting with it. In a DVE, untrusted third parties interact with state. They may choose to make changes or render state in a way which violates DVE rules. In a game DVE, this is *cheating*.

A client-server DVE retains central authoritative state against which shadow copies are reconciled, helping to mitigate many attacks. When a DVE is implemented as a P2P distributed system, more security challenges arise. There is no longer a single server or cluster with authoritative state. Instead, state authority is distributed, requiring more complex state change and reconciliation models, and providing additional opportunities for attackers.

We propose Carbon, a trusted auditing system for DVE's. Carbon helps to address security concerns particular to P2P DVE's, namely the modification of state in ways which violate DVE rules.

Section 2 provides an overview of relevant DVE research. Section 3 outlines the Carbon auditing system. Section 4 evaluates Carbon's attributes and performance. Section 5 presents conclusions and future work.

2. Related Work

Relevant DVE research falls into three categories: full P2P DVE frameworks, DVE topology and state management, and DVE security. We follow this with a brief discussion of DVE traffic patterns for those unfamiliar with the domain.

2.1 P2P DVE Frameworks

While we are unaware of P2P DVE's or frameworks in broad use, several are proposed in literature. Two of the most prominent are SimMud and the Colyseus framework.

SimMud[1] proposes a DVE framework based upon traditional P2P infrastructure. It uses the Pastry[2] DHT to store and retrieve key-value pairs and organize other overlays. The Pastry-based Scribe protocol[3] provides application-layer multicast, enabling pub-sub event distribution. This architecture relies upon the indifference of random strangers for correct operation. Authoritative state for each variable is stored at a master chosen by its Pastry ID. 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, or prevent state from correctly propagating to subscribers.

Colyseus[4] allows state to be stored at a state-appropriate master (such as the node which 'owns' the state), but leverages both traditional and range-query DHT's for indexing state location. State shadow copies can be created as needed, updated according to their access pattern. Colyseus' authors evaluate their framework by adapting Quake II to run on it. Like SimMud, the Colyseus framework emphasizes low latency and performance over security. For example, there are no allowances for detecting and addressing illegal state modification.

There are other P2P frameworks and research DVE's, but like SimMud and Colyseus, they tend to focus on core performance attributes such as consistency and latency rather than security.

2.2 Network Topology and State Distribution

DVE communication patterns significantly affect consistency and attack vulnerability. For example, a client-server DVE can enforce simulation rules at the server, minimizing illegal state updates. Or, DVE's in which all participants have copies of all state – a P2P DVE with full state replication – retain continuity of contact and universal knowledge, making detection of illegal state modification easier.

DVE's in which state is neither centralized nor universally propagated are more vulnerable to state attack. A typical example is region-based DVE's. The DVE participants are organized into regions based upon geographical coordinates. All participants in each region are aware of each others' state. A node entering and leaving a region is – from the perspective of other nodes in the region –similar to a node joining or leaving the DVE altogether. Without appropriate safeguards, an entering node can misrepresent its state to all other nodes in the region. In another attack, it can fork its state and be active in several regions simultaneously.

If the DVE uses third parties to transfer state between the authoritative state owner and shadow state subscribers, a number of other attacks are enabled. Most notably, state can be delayed or dropped, and in some cases modified or forged.

Vulnerability to all of these attacks is heavily influenced by the topology used for state storage and dissemination. The majority of solutions in current literature have state owners propagate state directly to shadow state subscribers, emphasizing topology to enable this. Some allow messages to be propagated via third parties, usually in a topology mirroring node location within the simulation.

VAST[5] is an area of interest (AoI) scheme for organizing DVE state exchange topology. A Voronoi diagram is constructed based upon node positions within the DVE. Each node maintains a relationship – 'enclosing neighbor,' 'boundary neighbor,' or simple neighbor – with a subset of other DVE nodes, based upon adjacency and coordinate distance between the nodes. This relationship determines which nodes are connected and exchanging updates.

[6] proposes using Delaunay Triangulation based upon node location to organize DVE topology. As a refinement, it suggests organizing highly proximate peers in the DVE into a cluster for more efficient message exchange. Delaunay Triangulation is the dual of a Voronoi diagram, so the overall topology is similar for both VAST and Delaunay Triangulation, modulo the cluster refinement.

The P2P Message Exchange Scheme proposed in [7] and [8] uses AoI to allow a node to partition nodes into three sets: 'active entities,' 'latent entities,' and others. Active entities are nodes close enough to have full fidelity in state information exchange. Latent entities are further away, but still close enough to need lower fidelity state updates. Others are nodes too far away from the node to require any information.

From a security perspective, the approaches listed in this section and in section 2.1 limit state update propagation to a dynamic subset of other DVE participants. DVE nodes come into and out of scope of other nodes, providing the opportunity for nodes to misrepresent their state upon introduction, or to fork their state, providing different state to non-overlapping sets of neighbors.

2.3 DVE Security

As mentioned earlier, P2P DVE's are vulnerable to a broader set of attacks than client-server DVE's. Significant research exists describing techniques to mitigate improper DVE state modification. In other words, preventing or detecting unauthorized state modification.

There are three main approaches: protecting DVE software and communications, distributing state ownership to disinterested third parties, and auditing schemes. Kabus et. al. [9] provide a good – although slightly out of date – overview of all three.

Protecting software integrity and communications is a trusted computing base (TCB) approach. Mobile Guards[10] are a good example. DVE software is changed to require a trusted software component – a *mobile guard* – be present and operational. Portions of the DVE 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 is guaranteed that its communications are unaltered, and that the DVE rules are being 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. This is an interesting approach, but it is predicated on the impossibility of an attacker cracking a mobile guard before it is updated. This core requirement is one of the issues dividing those who accept TCB's as sufficient and those who do not.

Distributing authoritative state ownership to disinterested third parties is another security technique. P2P DVE's such as SimMud, Colyseus use this method. A variant presented in IRS[11] allows state to be owned by the concerned party, but audits state update calculations by performing them at multiple untrusted nodes, then comparing the results. It assumed a disinterested third party has no motivation to break DVE rules in terms of state representation or updates, and that several disinterested parties are unlikely 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.

PeerReview[12] 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 *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, suspected, or exposed (bad). Audit frequency guidance is provided in terms of number of witnesses and the probability of illegal activity.

PeerReview is a flexible, decentralized, general system with strong security guarantees. It is designed to operate in the absence of any central authority, and under a variety of scenarios. Our proposed system is similar to PeerReview, but makes the simplifying assumption of a trusted audit authority. Consequently Carbon can provide appropriate DVE security with fewer preconditions. For example, our system requires participants to log their activities, but these logs don't need to be as rigorously protected as PeerReview logs. PeerReview is designed to operate in parallel with the system it protects, while Carbon is a parasitic system piggybacking on the host DVE's messaging. Also, Carbon has a lighter weight commit requirement, needing only periodic state commit rather than full log commit to its authority.

2.4 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 instant response to interactions. Many of these DVE's are fast-paced combat simulations, with real-time activity such as aiming and firing weapons, and chasing opponents. These applications are extremely latency sensitive, with latencies greater than 200 ms significantly degrading the experience[13][14].

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 (33 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. Again typical 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 [17], between 10 and 1164 kbps mean download bandwidth consumption, and between 13 and 74 kbps mean upload bandwidth consumption.

Since game DVE's are dramatically more popular than any other type of DVE, we limit our analysis to game DVE traffic patterns.

3. Carbon

Distributed Virtual Environments 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 types of 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 audits 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 a DVE uses to organize auditable information. In most cases enabling use of 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 components 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

DVE's run the gamut from virtual reality applications like Second Life to networked games like Quake III and World of Warcraft. Implementation techniques vary widely, as does nomenclature.

Table 1 introduces the terminology we will use to describe DVE concepts used by Carbon.

Table 1 – DVE Nomenclature

Symbol	Meaning
S_i^t	State at node i at time t
E^t	Event received at time t
ID_a	DVE identity for 'a'
$L_i^{t1,t2}$	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.

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.

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. $S_i^{t1+\epsilon} = ProcessEvent(S_i^{t1}, E_i^{t2})$. 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.

DVE state changes must be deterministic. This doesn't rule out choosing state changes randomly, but it does mean that given a random choice must induce a deterministic change, and the choice is itself an event.

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*. It 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, both state snapshots and events are opaque data blobs, stored as simple byte arrays.

The reporter exposes the interfaces described in Table 2 for the exclusive use of the DVE node.

Table 2 - 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
AuditRequest	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 a 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 audit client returns data to the DVE node:

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. 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.

Table 3 - 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 functionsThe only new interface is *CompleteAudit*, used by the DVE server to return audit results to the Carbon auditor, along with an optional 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. 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 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 calls *RequestAudit*. This call is authoritative since it is made by an auditor, and results in a log excerpt request for the specified node.

When the log excerpt is received and raised as a notice, the DVE component 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.

The example 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. We require state snapshots 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.

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 $M_{A\rightarrow C1}$ to send to the auditor. The DVE node connects to the appropriate DVE server, transmits the message, and calls *ReleaseNotice($M_{A\rightarrow C1}$)* 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($M_{A\rightarrow C1}$)*. 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 transmits 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* notification *(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. Evaluation

P2P DVE's as a class have more significant security vulnerabilities than client-server DVE's. State storage and modification is performed on untrusted peers. There is no guarantee any node is executing the prescribed code base. As a result, DVE nodes requires means to ensure correctness of other nodes behavior.

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 AoI.

Given a set of trusted audit nodes, the Carbon system allows a DVE to mitigate vulnerabilities related to misrepresentation of state, and to detect illegal modification of state. 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.

The remainder of this section evaluates two aspects of Carbon. First, mitigation against incorrect state modification. Second, the overhead involved in using Carbon.

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 typically perform similar activities to one another, with similar levels of trust, ideally none. In some architectures, a node may be granted additional responsibilities for coordinating DVE activities, but such responsibilities are typically based upon node resources, not 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 ensure avatar integrity and correctness. It proved 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 wishes to violate DVE rules by modifying her avatar to have a billion 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 the claim can be verified. 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 that money is suspect, its source in the DVE can also be audited, and so on.

Carbon can detect the following existing attacks:

1. Illegal avatar activity, such as teleport hacks and speed hacks violating DVE rules for movement.
2. Avatar edits, such as changing the resources associated with an avatar. Network game attacks in this category include 'god mode,' duplication attacks, and weapon edits.

Exhaustive auditing via Carbon reliably detects these attacks and has reasonable overhead, as discussed in section 4.2. If less expensive auditing is desired, DVE's can leverage participant affinity for their avatar, threatening to punish any cheaters. The DVE adjusts punishment and audit frequency to achieve deterrence.

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 Overhead

Integrating Carbon requires relatively little effort for an existing DVE. However, Carbon does introduce significant network traffic and processing overhead.

DVE operation can be characterized by four main activities:

1. Abstract simulation. Evaluating state change and events. This is typically a lightweight activity.
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.
3. Persistence. Storing DVE data, for example saving avatar state for later retrieval. Usually a very lightweight activity.
4. Network traffic. DVE's must exchange messages to determine state changes and to refresh shadow copies of state. In a P2P DVE, this traffic is typically the traffic required to describe a state change traffic multiplied by the shadow copies.

Carbon auditing doesn't influence abstract simulation and rendering at DVE clients, but it does increase persistence and network traffic workload. Carbon requires simulation data be retained in two places. The reporter on each DVE node maintains a local log of event and state snapshots. The auditor authoritatively stores state snapshots. Total storage consumption is significant, but can be reduced by flushing data past an assigned audit time horizon.

The bulk of transmitted Carbon data comes from state snapshots and audit log excerpts destined for auditors. The scale of this activity varies according to DVE, but it is intuitively guaranteed to require at least as much network bandwidth as the packet payloads an efficient DVE uses transmitting state data.

For illustration, suppose we have a DVE with the following attributes:

Table 4 - Sample DVE attributes

Function	Variable
State snapshot in kilobytes	$\|S_i\|$
Average outgoing event rate	$\bar{R}(E_i)$
Average shadow state copies	Q
Commit interval	CI

$|S_i|$ is size in kilobytes for a node's full state description. This typically includes avatar position and attributes, and the state of objects being tracked by that node. $\bar{R}(E_i)$ is the average traffic in kbps required to describe a node's state transitions to a neighbor. Q is the average neighbors receiving shadow state updates, and is one less than the average clique size (number of nodes in mutual interaction range). Commit interval CI is the interval in seconds between DVE node calls to *Commit()* for committing state snapshots to the auditor.

Incoming and outgoing non-audit bandwidth for a P2P DVE must be at least $\bar{R}(E_i) \times Q$: Each client transmits Q copies of its state changes, one to each clique member besides itself. Each client receives one copy of state updates from each of its Q fellow clique members.

Per-node bandwidth required for auditing *Commit* is $|S_i|/CI$ kBps.

Suppose p is probability of auditing the period between two state snapshots. Per-node reporter network overhead is $(|S_i|/CI) + p \times \bar{R}(E_i) \times (Q + 1)$. We multiply the event data rate by $(Q + 1)$ instead

of Q because each node needs to provide the auditor with events it originates as well as those it receives.

To help put the overhead in perspective, we use World of Warcraft (WoW) traffic models from [16] coupled with our own measurements obtained using WireShark and WoW version 3.1. We choose WoW because of its popularity and hence probability of benefiting from a reliable P2P DVE approach. WoW clients requires on average 2.1 kbps upload and 6.9 kbps download bandwidth. It uses a command / state response model. We verified this by measuring bandwidth consumed by both an actor node and a node closely observing their acts. Hence, we consider the state change traffic to be 6.9 kbps per single copy of state changes transmitted, including network overhead.

As mentioned in section 2.4, most game DVE packets have a very small or empty payload. 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 packets were received, with a total size of 284,246 bytes. Our per-packet transport overhead for TCP and proxy 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 in communications. This tells us the numbers from [16] provide conservative values, and are therefore a good benchmark for our purposes.

Audit log extracts can be efficiently transmitted, as they are many MTU's in size. This drops transmission overhead from about 60% to approximately 5%. Translating this to aggregate event data numbers, event summaries for events at a given node consume $\bar{R}(E_i) = 2.9$ kbps compared to the 6.9 kbps required for the DVE's operational traffic pattern.

Based on our measurements, a full state snapshot as received from the server on initialization depends heavily upon the area entered. We measured values between 26 kB for a quiet area to 125 kB for a busy one. For this example, we set $|S_i|$ = 62.5 kB, roughly halfway between the two extremes. We assume Q is 10.

Average bandwidth sent and received by a P2P DVE node in our scenario – excluding audit-related activities – is (6.9 * 10) inbound + (6.9 * 10) outbound = 138 kbps. Suppose we choose a *Commit* interval $CI = 15$ minutes. Then we have (62.5*8 / 900) = 0.56 kbps for state snapshots, and p * (2.9 * (10 + 1)) = p * 31.9 kbps for reporting log excerpts, for a total of 32.5 kbps for full auditing of all state transitions. In other words, full auditing requires 24% network bandwidth overhead compared to the DVE node's normal operating requirements.

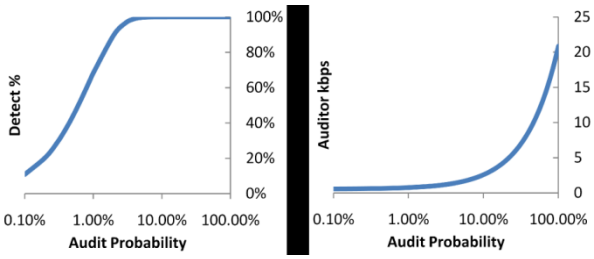

Figure 1 - Detect chance and overhead by audit chance

Auditing lacks the immediacy of a client-server approach to the DVE security, but it also has advantages.

1. Since each audit simulates a single node's snapshot of the world, we have proof by example that it can be performed by a single node. Auditing the entire DVE is decomposable into auditing individual nodes. This means auditing scales linearly with the number of front ends compared to sub-linear scale of centralized DVE servers for client-server DVE's.
2. The fraction of state intervals audited can be scaled down according to the risk tolerance and participant avatar affinity, reducing audit cost. In our example the DVE could audit a small fraction of sessions (such as $p = 0.01$). Someone who broke the rules only once an hour would have a nearly 70% chance of getting caught during their 100-play-hour lifetime. If $p = 0.05$ the probability of being caught and punished rises to 99.7%. Many game DVE's have avatar lifetimes measured in *thousands* of hours, not hundreds.

The second point underscores the efficacy of random auditing for catching cheaters *at least once in their expected avatar lifetime*. Figure 1 shows the probability of catching a cheating avatar sometime in their lifetime for a given random audit fraction p. In our example, lifetime cheater lifetime detection ramps up to 99.9% with 7% random auditing.

Figure 1 also shows the linear per-node audit bandwidth overhead for a given value of p. Bandwidth overhead is linear with audit fraction. Audit selection based on state examination and adversary-proposed audits should have a better success rate, but this result shows even random auditing can act as a deterrent for habitual rule violators.

Storage overhead for client logs and for the central state snapshots would not be prohibitive for modern computers. The storage required at a node for a given time window is roughly the same as the auditing network overhead per node times the time period. In our example, a full week of 24-hour audit storage for a DVE client would consume no more than (7 * 24 * 60 * 60) s * 32.5 kbps = 2.5 GB of audit client storage.

Auditor storage required to track state snapshots for each node for the same time window would be 42 MB.

5. Conclusions

P2P DVE's are an active research area, and so far haven't been broadly deployed. One requirement for broad deployment is a good security solution to mitigate cheating, providing security guarantees similar to those available in DVE's deployed today.

We introduced Carbon, a trusted auditing system. Carbon consists of a per-node audit client component called a reporter, and at least one trusted auditor run as part of a trusted DVE component. DVE's using Carbon can ensure that DVE state transitions performed by DVE participants are at least plausible, and that those participants are not forking their state or performing illegal state transitions such as teleport and speed hacks.

We showed that Carbon overhead is significant but not untenable. In a scenario modeled after a P2P World of Warcraft DVE, we showed Carbon requires on average only 24% additional network traffic to perform exhaustive auditing. We further showed that for DVEs with strong participant-avatar affinity (such as World of Warcraft), auditing only a fraction of participant activity – such as 1% – still provides a good chance of catching state transition violations, deterring would-be cheaters and hackers. We also showed that carbon auditing can be scaled linearly with the number of auditor resources applied.

References

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

[2] 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.

[3] 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.

[4] Ashwin Bharambe, Jeffrey Pang, and Srinivasan Seshan, "Colyseus: A Distributed Architecture for Online Multiplayer Games," in *NSDI '06 Proc.*

[5] Shun-Yun Hu and Guan-Ming Liao, "Scalable peer-to-peer networked virtual environment," in *NetGames '04 Proc.*, pp. 129-133.

[6] Ernst Biersack, Christophe Diot Matteo Varvello, "Dynamic Clustering in Delaunay-Based P2P Networked Virtual Environments," in *NetGames '07 Proc.*, pp. 105-110.

[7] Yoshihiro Kawahara, Tomonori Aoyama, and Hiroyuki Morikawa, "A Peer-to-Peer Message Exchange Scheme for Large-Scale Networked Virtual Environments," *Telecommunication Systems*, vol. 25, pp. 353-370, 2004.

[8] Nobutaka Matsumoto, Yoshihiro Kawahara, Hiroyuki Morikawa, and Tomonori Aoyama, "A scalable and low delay communication scheme for networked virtual environments," in *Global Telecommunications Conference Workshops, 2004*, pp. 529-535.

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

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

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

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

[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," , 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.