

# Design and Implementation of Generalized Functional Monitoring

Kelly Heffner                      Gregory Malecha

May 16, 2009

## Abstract

Distributed monitoring of values is a common task in many distributed systems including sensor networks and online games. In this work we develop a generalized framework for functional monitoring of values and show how it can be applied in the context of online games. We implement a library for general functional monitoring and consider the tradeoff between network usage and accuracy with different types of values.

## 1 Introduction

Massively Multi-player Online Role-Playing Games (otherwise known as MMORPGs) are collaborative video games that support simultaneous game-play by hundreds or thousands of players in the same online world [2]. During any typical moment in an MMORPG world, hundreds of players are engaging in battles with monsters like the portrayal in Figure 1. Multiple players are dealing damage, measured in numerical *hit points* to a monster. Each player is doing damage in different amounts and at different times independent of the other players – all of the players are concerned with getting the hit points of the monster down to zero, at which point they win the battle.

Throughout such a battle, the game server is tracking the value of countless important statistics: the hit points of each player and monster, changing statistics about each player (strength, dexterity, or charisma, for example), experience points for the players, and how much money each player has, just to name a few. The players can affect these statistics and communicate the updates to the server. This is just one battle though – there are hundreds of such battles going on at any moment on the server.

Additionally, there is shared game state of types that are not just simple integer values. Non-player characters, or NPCs, typically interact with the players through dialog, exchange of items, and movement in the game world. Some NPCs have an AI which is built on a state machine [1], such that interactions from all of the players cause state transitions.

Communication of all of these changes from the client to server, and of updates from the server to the client, requires bandwidth to ensure that the game-play runs smoothly.



Figure 1: A portrayal of a typical battle in an MMORPG.

However, game makers need to try to minimize bandwidth for two reasons: The game needs to be marketable to consumers, who may or may not have a high speed connection [4], and the game companies need to minimize the bandwidth utilized by their game server farms, to minimize their overhead costs [7].

Because these games are networked, it is a common for players to not have expectations of perfect real-time responsiveness from their game. Some delay induced by network latency (referred to as “lag”) is expected, as are as minor inconsistencies (referred to as “glitching”) caused by the inherent race conditions on the shared server state.

## 1.1 Contributions

In this work we study the properties of functional monitoring in the standard setting of tracking numbers and generalize this mechanism to arbitrary state machines. Specifically, we

- Generalize functional monitoring to arbitrary values by describing a minimal set of operations necessary for monitoring of a value (Section 3).
- Describe desirable properties of value interfaces and the guarantees that those properties provide (Section 3.2)
- Provide an implementation of our generalized functional monitoring and provide empirical results (Section 5.2).

We begin with a review of functional monitoring and current protocols (Section 2) before addressing each of our contributions. We conclude by considering extensions of our work (Section 6).

## 2 Background

### 2.1 Client-Server Architecture for Online Games

Multi-player online game networks are typically composed of many clients and fewer servers.

**Servers** are the authority on the game state. They collect update messages from clients and NPC AI, and combine them to update the official world state. Due to the heavy connectivity, storage, and computational demands on servers, users of an MMORPG are normally into fixed sets that can be processed independently.

**Clients** traditionally handle player aspects of the game, such as graphical rendering, and rely on the server to tell them about the world, such as the positions and status of other entities. However, most popular clients today, such as World of Warcraft [3], try to reduce server processing by guessing updates with a model, and periodically synchronizing their view of the world with the servers. This decreases server congestion at the cost of possible glitching.

### 2.2 Functional Monitoring

The scenario described above can be abstracted into an instance of a class of problems known as *functional monitoring* problems [8]. A functional monitoring problem is composed of  $k$  client machines tracking their own inputs and communicating them to a central coordinator. The coordinator monitors a function,  $f$ , over the entirety of those inputs at all times  $t$ . In our motivating example, the coordinator is the central game server, and the function  $f$  is the amount of damage done to a monster at times  $t$  by all of the players. The coordinator is required to alert (by killing the monster) when the total damage done meets the threshold  $\tau$  (the total health of the monster).

In order to reduce the amount of communication between the clients and central coordinator, we are interested in the approximate version of this problem. The coordinator must alert when the function meets the threshold,  $f \geq \tau$ , and not alert when the function is beneath a certain fraction of the threshold,  $f \leq (1 - \epsilon)\tau$ . The four-tuple  $(k, f, \tau, \epsilon)$  defines our approximate distributed functional monitoring problem.

### 2.3 Communication Strategies

In the work by Cormode [5], the authors describe three randomized communication protocols, two of which we have adapted for our purposes.

**Coin-Toss** Clients accumulate their inputs locally. Each time a client has an input, a coin flip is done to determine if the client should send the accumulated input. In the standard formulation, the communication probability is held constant; though this is not a strict requirement.

**Global** The coordinating server computes and communicates a significance threshold  $\tau$  to each of the clients. Clients accumulate their inputs locally. Each time a client has an

input, the client tests the new accumulation to see if it surpasses the threshold  $\tau$ ; if it does, the client sends a notification to the server. When the server receives a certain number of notifications, it signals the end of round to all of the clients. Clients all respond to the end of round message by communicating their accumulated inputs, and the server merges all inputs from all of the clients once they have been received.

**Local** Each client computes a model of its inputs over time, and communicate that to the coordinating server. The server updates the shared state according to the models it receives from the clients. Each time a client has an input, the client accumulates its deviation from the model communicated to the server. When a client has deviated from its model significantly, the client communicates its deviation to the server, computes a new model for its inputs, and communicates that new model to the server.

### 3 Generalizing Values

As mentioned in Section 1, there are instances in which the shared value has a more complicated form than simple integers.

#### 3.1 A Generic Value Interface

By abstracting the algorithms, we can distill the pieces necessary for monitoring general values. We begin by abstracting the type of the value being monitored ( $\nu$ ). However, since the coordination protocols operate directly on the value, each protocol requires additional operations over these value. We consider the coin-toss and global protocols and the abstract operations needed to support them.

In addition to the type of the value being monitored, the coin-toss algorithm must be abstracted with respect to the encoding of the difference ( $\delta$ ). In the previous example of integers, this difference was simply the difference (subtraction) that the client is responsible for. With these types abstracted, the server requires a merging function to combine values and differences into values. Similarly, the client needs a function to combine differences into a difference when an update is not sent. With these four pieces, we can implement a generic coin-toss implementation. The types required are:

$$\text{Observable } \nu \delta = \left\{ \begin{array}{ll} \text{mergeValue} :: \delta \rightarrow \nu \rightarrow \nu & (\text{server}) \\ \text{mergeDiff} :: \delta \rightarrow \delta \rightarrow \delta & (\text{client}) \\ \text{nullDiff} :: \delta & (\text{client}) \end{array} \right\}$$

The only piece that we have not considered in the above specification is `nullDiff` the unit for  $\delta$  satisfying `mergeDiff nullDiff x = x` and `mergeValue x nullDiff = x`. We include this as a conceptual convenience so that a value can be stored when no difference has been updated.

Next, consider the global protocol. The requirements for the coin-toss algorithm serve as an reasonable start; however, it is not sufficient. In addition to the value and difference,

the server must communicate a threshold ( $\tau$ ) which decides whether a given difference is significant and the appropriate eliminator for answering this question. This results in the following signature:

$$\text{Observable } \nu \delta \tau = \{$$

<code>mergeValue</code>	:: $\delta \rightarrow \nu \rightarrow \nu$	<i>(server)</i>
<code>mergeDiff</code>	:: $\delta \rightarrow \delta \rightarrow \delta$	<i>(client)</i>
<code>nullDiff</code>	:: $\delta$	<i>(client)</i>
<code>threshold</code>	:: $\delta \rightarrow \tau \rightarrow \mathbf{Bool}$	<i>(client)</i>

$$\}$$

This simple definition leaves one remaining point open; the global algorithm conceptually merges all deltas for a round simultaneously. In cases in which order matters (see Section 3.2) the merge order matters, and in some cases, merges might not be additive; for example, taking the max or min delta over a round since clients did not necessarily observe the updates of others (we discuss the broadcast problem in Section 4). Therefore, we generalize the `mergeValue` function to accept a set of  $\delta$ s. Our final requirements for observability in the global algorithm is therefore given by:

$$\text{Observable } \nu \delta \tau = \{$$

<code>mergeValue</code>	:: $\delta \rightarrow \nu \rightarrow \nu$	<i>(specification)</i>
<code>mergeValues</code>	:: $[\delta] \rightarrow \nu \rightarrow \nu$	<i>(server)</i>
<code>mergeDiff</code>	:: $\delta \rightarrow \delta \rightarrow \delta$	<i>(client)</i>
<code>nullDiff</code>	:: $\delta$	<i>(client)</i>
<code>threshold</code>	:: $\delta \rightarrow \tau \rightarrow \mathbf{Bool}$	<i>(client)</i>

$$\}$$

We will explore several strategies for converting a `mergeValue` function into a `mergeValues` function in a  $\delta$  agnostic manner in Section 3.3. However, we believe that a single generic implementation is not sufficiently general for all purposes.

## 3.2 Interface Properties

Thus far, we have considered a generalized interface for the value-monitoring from a type-oriented perspective; however, we have not yet considered the properties that these functions should have. In this section we consider several properties that these operations should strive for. We start by considering several simple, single-client properties, and then consider multi-client properties and their effects on the monitoring results.

Several local properties are desirable for consistency when working with differences and tolerances. We have already discussed the properties of `mergeDiff` and `nullDiff`; mainly that `nullDiff` should at least be a left-identity for `mergeDiff` and a right-identity for `mergeValue`.

$$\forall d. \text{mergeDiff } \text{nullDiff } d = d \quad \forall v. \text{mergeValue } v \text{ nullDiff } = v$$

This addresses the practical notion that doing nothing should correspond to no change in the server. On the client side, it further makes sense that

$$\forall t. \text{threshold } \text{nullDiff } t = \mathbf{False}$$

Since a difference which will have no effect on the value. Note that without this requirement, the global algorithm will continuously send threshold messages since the delta value is set to this when no difference has been generated since the last send.

At a global level, several properties are required in order to guarantee consistency of the tracked value. Before proceeding we give several technical definitions. The *ideal value* at time  $t$  ( $v_t$ ) is the result of merging the initial value with each  $\delta$  as it is generated by each client. This corresponds to sending the difference on every update with an infinitely fast network. The *perceived value* of the coordinator at time  $t$  ( $\bar{v}_t$ ) is the value that the server uses in its thresholding comparison.

Consider first the accuracy of the coin-toss protocol. What conditions are necessary in order to guarantee that the server sees an accurate picture of the world when it is synchronized with the clients? The coin-toss protocol has two interesting features, each of which requires a property on the interface. The first of these features is combining differences. To maintain consistence, this requires the distributivity of `mergeValue` over `mergeDiff`. Formally, this is defined as:

$$\forall v d_1 d_2. (\text{mergeValue } d_2) \circ (\text{mergeValue } d_1) v = \text{mergeValue } (\text{mergeDiff } d_1 d_2) v,$$

The second property, also a product of delayed transmission is the lack of global ordering. Consider the following example with two clients: Client 2 sends an update before client 1 but the update sent by client 1 includes a difference generated before the update of client 2. In order for the resulting value to be the same as the ideal value, we require that the `mergeValue` function is commutative, i.e.

$$\forall v d_1 d_2. (\text{mergeValue } d_1) \circ (\text{mergeValue } d_2) v = (\text{mergeValue } d_2) \circ (\text{mergeValue } d_1) v$$

Given these two properties, we can state the following error-consistency property for the Coin-Toss protocol.

**Theorem 1** (Coin-Toss Ordering/Max Error). *Let  $v_0 \in \nu$  be the initial value of the coordinator,  $\langle d_i \rangle^k \in (\delta^*)^k$  be the sequence of differences produced by  $k$  clients. At any time  $t$ , the perceived value of the coordinator ( $\bar{v}_t$ ) merged with the differences of the clients is equal the ideal value ( $v_t$ ).*

*Proof.* Follows from the distributivity of `mergeDiff` and `mergeValue`. □

The following corollary defines when the server knows the ideal value.

**Corollary 1** (Coin-Toss Ideal Value). *Let  $v_0 \in \nu$  be the initial value of the coordinator,  $\langle d_i \rangle^k \in (\delta^*)^k$  be the sequence of differences produced by  $k$  clients. At any time  $t$  in which no client has generated a difference since last updating with the server, the  $\bar{v}_t = v_t$ .*

*Proof.* Follows immediately from Theorem 1 and that `nullDiff` is the left identity for `mergeValue`. □

A similar property can be derived for the global algorithm since global simply imposes determinism on when updates are sent. Since the correctness of the value is defined with respect to `mergeValue` and the system performs merging using `mergeValues`, we require that these be consistent; namely:

$$\forall v d. \text{mergeValue } d \ v = \text{mergeValues } \{d\} \ v$$

Again, we require properties for `mergeDiff` and `mergeValue` that allow us to merge  $\delta$ s without losing information. For the global algorithm, this requires the following properties.

$$\begin{aligned} \forall v d_1 d_2. (\text{mergeValue } d_2) \circ (\text{mergeValue } d_1) \ v &= \text{mergeValues } \{d_1, d_2\} \ v \\ \forall v d_1 d_2. \text{mergeValues } \{d_1, d_2\} \ v &= \text{mergeValues } \{\text{mergeDiff } d_1 \ d_2\} \ v \end{aligned}$$

Together; these imply that `mergeValues` can be implemented as:

$$\begin{aligned} \text{mergeValues } v \ [] &= \text{nullDiff} \\ \text{mergeValues } v \ (ds_1 \text{ concat } ds_2) &= \text{mergeValues } ds_2 \ (\text{mergeValues } ds_1 \ v) \end{aligned}$$

Given the above properties, we can define and prove the correctness and error bounds of the global algorithm.

**Theorem 2** (Global Ordering/Max Error). *Let  $v_0 \in \nu$  be the initial value of the coordinator,  $\langle d_i \rangle^k \in (\delta^*)^k$  be the sequence of differences produced by  $k$  clients. At any time  $t$ , the perceived value of the coordinator ( $\bar{v}_t$ ) merged with the differences of the clients is equal the ideal value ( $v_t$ ).*

*Proof.* Follows from the properties above. □

**Corollary 2** (Global Ideal Value). *Let  $v_0 \in \nu$  be the initial value of the coordinator,  $\langle d_i \rangle^k \in (\delta^*)^k$  be the sequence of differences produced by  $k$  clients. At the round ending at  $t$ , the perceived value of the coordinator ( $\bar{v}_t$ ) is equal to  $v_t$ .*

*Proof.* Follows from Theorem 2 and that `nullDiff` is the left identity for `mergeValue` since at time  $t$ , the value stored at the client that was not sent was `nullDiff`. □

This corollary is stronger than that of the coin-toss algorithm because it yields a way for determining when the server is synchronized; namely immediately after it has collected all of the differences.<sup>1</sup>

### 3.3 Case Study: State Machines

In Section 3.1, we defined the types and operations necessary for making a value observable. We now consider applying our framework to the problem of monitoring a state machine. Of particular interest is that the transitions in the state machine are often not commutative. Throughout the remainder of the section we consider different ways to address this problem.

---

<sup>1</sup>We ignore issues of communication latency in our analysis.

**Sequence of Inputs:** A naive choice of  $\nu$  and  $\delta$  is to let  $\nu$  be the states and  $\delta$  the set of finite sequences of input symbols. While suitable in the single client case, this choice lacks the desired commutativity properties described in Section 3.2. While the drawbacks of this are immediately apparent, the approach may be sufficient for many cases. For example, when clients are acting based on their own state which may already differ from the perceived state.

For the coin-toss protocol, the `mergeValue` function is fairly standard. Since only a single input sequence is seen at a time, the most reasonable approach is to simply follow the differences in order to determine the new state. The round-based approach of the global algorithm, however, lends itself to more variety. Some potentially meaningful implementations of `mergeValues` are:

**Queuing** Use any standard queue merging technique, such as round-robin. This merge technique works well in situations in which the clients update rates are predictable; for example, in turn-based games.

**Probabilistic** Construct a probability distribution on the final states based on the potential interleavings and select a final state at random according to the distribution—alternatively, the most likely state could be picked arbitrarily. Though constructing the probability distribution for a DFA with a large set of symbols could be computationally expensive, in some situations, this would be an easy technique for adding some variation to an NPC behavior.

**Hand-crafted** By exploiting the contextual meaning of the state machine, the final state can be chosen with various properties. For example a pessimistic or optimistic merging would find the worst or best final states for the clients.

For example, using our DFA from Figure 2, a good `mergeValue` function might be one that orders the interleaving such that as many  $b$  input symbols appear before the first occurrence of an  $a$  input symbol as possible.

**Timestamping Differences:** If achieving the ideal state is essential in the meaning of the shared, we can augment the previous approach with a shared clock and encode timestamps in the  $\delta$ s along with the diffs themselves. As long as we can reconstruct the global ordering we can merge using the specification definition of ideal value. As before, `mergeDiff` can simply concatenate the two differences and `nullDiff` is simply the empty sequence.

In the coin-toss case, we need to construct a commutative `mergeValue` function; however, our differences carrying timestamps do not, alone, help with this since the value itself does not have any notion of time encoded in it. Naively, in order to make `mergeValue` commutative, we must store the entire history of the value; something which is likely unacceptable in a real-world application. We note, however, that if we can determine a time  $t$  such that we can guarantee that we will never see a timestamp before  $t$ , then we can roll the state value forward to that point and forget the history prior to  $t$ . The existential nature of Corollary 1 makes this slightly problematic within our framework; however, arbitrarily cutting off at some point

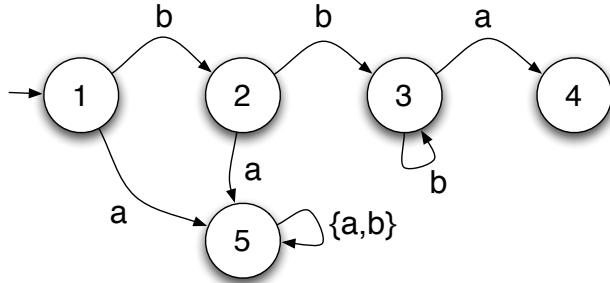


Figure 2: A simple FSM with a non-commutative set of transitions.

or tracking the clients who could send updates and rolling forward to the minimum of the maximum timestamps sent by any of them is sufficient.

Based on Corollary 2, this is much easier to implement in the context of the global algorithm since invocations of `mergeValues` have an implicit epoch property that timestamps prior to the latest timestamp in the set will not occur again.<sup>2</sup>

**Final State:** In some circumstances, the specific interleaving of the input symbols may not be critical to the process of computing a coherent value for the shared DFA state. In our example DFA (Figure 2), we could change say that talk is cheap and flattering the NPC must be done at least twice (transition *b*) while a bag of gold only needs to be given once (transition *a*).

If this is the case, we can summarize sequences of differences by the resulting state. In this case, `mergeDiff` implements following a transition in the DFA and `mergeValue` must reconcile the list of final states, which can be done in various ways. Simple merge operations are majority votes and “best” and “worst” final states. One could also use contextual information about the DFA to implement a more sophisticated state `mergeValue`. In our example, a more sophisticated `mergeValue` function would be:

If the state list contains,	go to state:
at least one 3 and at least one 5	4
at least one 5 and no state 3's	5
at least one 3 and no 5's	3
at least one 2 and no state 5's	2

## 4 Pushing the Authoritative View

The functional monitoring problem deals only with maintaining the value at the coordinator; however, in some applications, this is not sufficient. For example, in the context of an MMORPG, while the server must maintain an accurate picture of the authoritative value,

<sup>2</sup>Again we are ignoring the effects of latency

clients must be able to view the value in order to update displays and possibly change the local computation model. Additionally, if the server is tracking values for which the `mergeValue` function is non-commutative, the server must be able to communicate its state back, to correct any clients that have diverged from the authoritative path.

Pushing values from a coordinator has been addressed in previous work [8, 6]. Protocols for addressing push are similar to the monitoring algorithms with the exception that the clients do not attempt to merge values because there is only one writer, the authoritative server. In these protocols, client updates are processed by the server and disseminated when conditions are met.

**Coin-Toss** The coin-toss push protocol flips a (possibly biased) coin each time that it receives an update event. If the coin is heads, then the new value is sent to clients. In order to avoid bursts of communication in a unicast model, the server can opt to reflip the coin for each client; possibly with different probability based on characteristics of the client.

**Dependent** In the dependent push protocol, each client communicates its error tolerance to the server. The server pushes a new value to a client when the difference between the last value sent to that client and the current value is significant by some threshold. This is a generalization of the algorithm proposed by Yi and Zhang [8]. The primary difference between this and the global algorithm for functional monitoring is that, since there is only 1 target, there does not need to be a notion of rounds.

**Model** In the model push protocol, the server communicates a model of the updates over time, analogous to the local algorithm. Also like the local algorithm, the server communicates significant deviations and model evolution to the clients. The model is essentially the derivative of the published value with respect to time.

## 5 Empirical Analysis

In this section we give a brief overview of our functional monitoring system, *FunkyMon*. We use our implementation to analyze the tradeoff between accuracy and network resources.

### 5.1 Implementation

We have designed and implemented a functional monitoring and value-notification framework, *FunkyMon*, to gather experimental results.<sup>3</sup> The system design is based on publish-subscribe interface and supports servers hosting multiple values to watch of different types. The high-level architecture of the system is given in Figure 3.

*FunkyMon* is implemented as a in Haskell library and includes implementations of both coin-toss and global publishing protocols as well as coin-toss and threshold-based monitoring protocols. Selection of publishing and monitoring protocols are independent allowing clients

---

<sup>3</sup>Full code available at: <http://people.harvard.edu/~gmalecha/projects/FunkyMon>

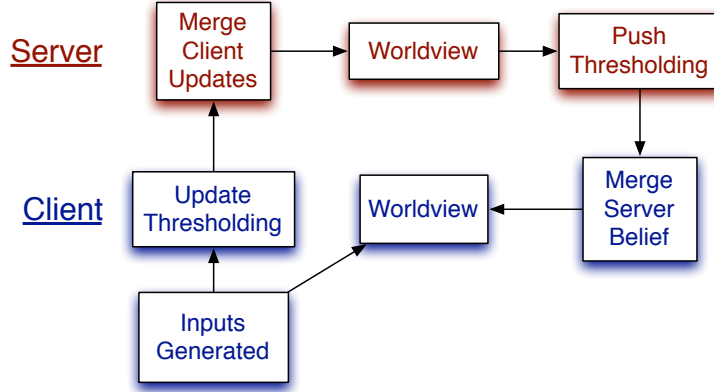


Figure 3: *FunkyMon* architecture.

to mix-and-match based on the needs of their systems. Each of the protocols is parameterized by the *Observable* operations given in Section 3.1.

## 5.2 Evaluation

One of the goals of this work is to understand the relationship between bandwidth usage and accuracy for distributed shared state. When evaluating bandwidth, we are concerned with the distribution of network communication over time – therefore we look at the total amount of data communicated, as well as peak bandwidth usage.

In the context of an MMORPG, errors can be broken down into three categories, in order of increasing severity:

**Lag** This is a more tolerable error than most; it is simply a difference in the time. When the ideal value reaches  $n$  and when the perceived value reaches  $n$  at the server. Normally, lag is caused by network latency, but our algorithms sometimes induce lag-like effects due to delaying updates.

**Glitching** This is a kind of error we would see from any non-commutative operations. The client acts on its perceived state which has diverged from the authoritative state. When the server updates this client to the authoritative state, the client will observe a potentially unnatural shift in state.

**Incorrectness** Due to a change in the interleaving of diffs from the true temporal progression, the authoritative state has diverged from the ideal state. This is symptom of non-commutative operations.

### 5.2.1 Integer Monitoring: Accuracy & Network

To put our results in perspective, we begin by comparing the integer-monitoring versions of the coin-toss protocol. We collected packet counts and server error-over-time measurements

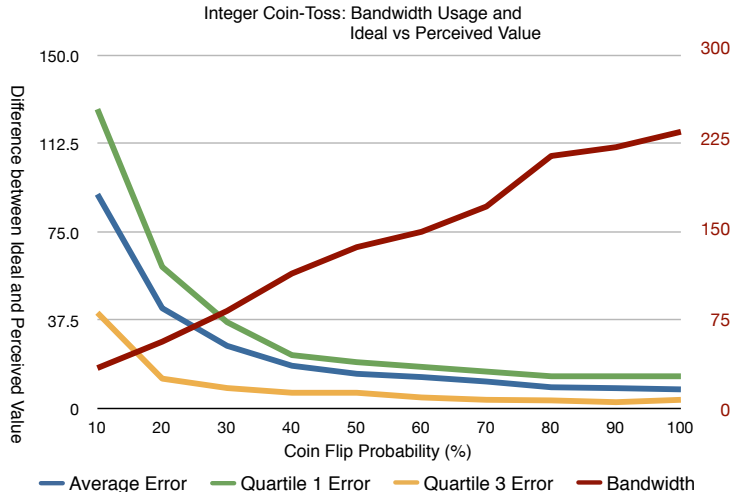


Figure 4: Integer-monitoring results for the coin-toss communication protocol.

for varying coin-toss probabilities. The results are summarized in Figure 4. Note that correctness errors do not occur in these experiments because addition is commutative.

As should be expected, the average difference between the ideal and perceived values reduces as the clients update the server more often. And of course, these more frequent updates occur at the cost of more bandwidth. It is interesting to note that beyond 70% transmission rate, the error is approximately stable while the bandwidth is still increasing. This suggests that, in practice, increasing transmission rates suffers from diminishing marginal returns.

### 5.2.2 Monitoring State Machines

In Section 3.3 we discussed several approaches to monitoring state machines within our framework. As with monitoring integers, we must define a measure for distance between the the perceived state and the ideal state. Our measure for difference is length of the shortest undirected path between the two nodes. This is naturally analogous to the difference measure used for integers and avoids the problem for infinite distances which are common in state machines.

For comparisons, we generated a random, connected FSM with 50 states and 3 input symbols. Input symbols were four times more likely to produce self-loops than go to any other single node. The undirected diameter of the FSM, and therefore maximum error, is 5. Each test was run for approximately a minute; we show results for a random window of approximately 5 seconds. The remainder of the time was roughly the same.

The nature of the FSM make these results fairly noisy. The structure of the FSM and the fact that there are three input symbols shrinks considerably the distances in the state machine making our error measure strange. Using a more semantic notion of states and differences might address this.

For the timestamping combination protocol, points at which the server error was 0 only occur in the coin-toss model. We believe that this is the result of network latency. Our ideal

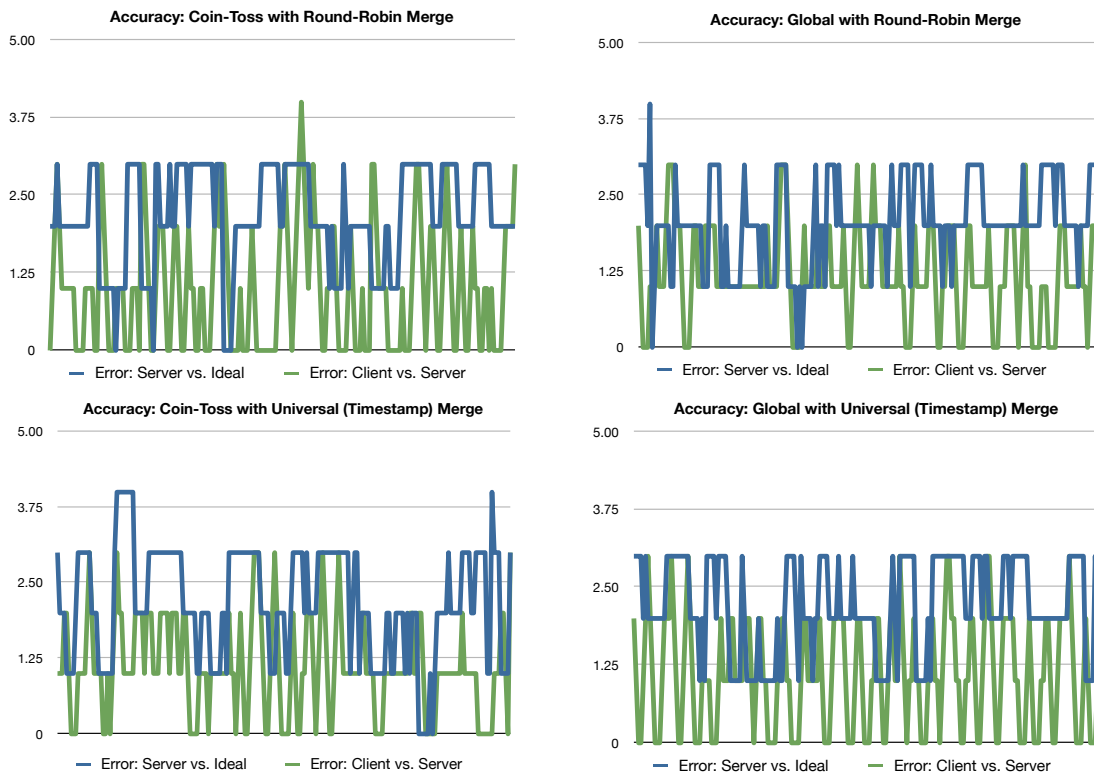


Figure 5: FSM-monitoring results for coin-toss and global protocols..

value is computed absolutely in time, so in the time between sending the round-ending and accumulating all of the results another change was generated. This explains why the server comes back to 1 at fairly regular intervals in the beginning of the graph.

## 6 Conclusions & Future Work

In this work we described a framework for functional monitoring of arbitrary values based on a set of types and operations. We derived properties that these functions must have in order to make certain guarantees about the accuracy of the tracked value over time. Using our framework we showed several ways that it can be applied to tracking arbitrary state machines. We provide an implementation of our framework and gave experimental results comparing various protocols.

Although the experimental results of this paper show positive indications that our value monitoring techniques can yield a savings in bandwidth usage, there remains a substantial amount of future work. One extension is to consider how functional monitoring works when multiple values are being tracked between the same client and server. Since real networks have minimum packet sizes, combining multiple updates to fill a single packet is a logical technique since there is no added bandwidth cost. The simple cost model which we have adopted in this paper is not sufficient for capturing this; but a more realistic model should be able to by measuring the utility of the bandwidth.

Our comparisons were restricted to the coin-toss and global protocols; however, the local protocol is useful in many circumstances. An analysis similar to the ones in Sections 3.1-3.2 is needed to understand what generic operations are essential and what can be guaranteed in a more general setting.

Finally, our implementation is simplified by using constant transmission probability (in the coin-toss protocol) and a constant threshold (in the global protocol). However, adapting these over time to compensate for increased or decreased importance or scarce bandwidth due to other monitoring efforts could provide a more efficient use of bandwidth.

## References

- [1] Wikipedia: Game artificial intelligence. [http://en.wikipedia.org/wiki/Game\\_artificial\\_intelligence](http://en.wikipedia.org/wiki/Game_artificial_intelligence).
- [2] Wikipedia: Massively multiplayer online game. [http://en.wikipedia.org/wiki/Massively\\_multiplayer\\_online\\_game](http://en.wikipedia.org/wiki/Massively_multiplayer_online_game).
- [3] World of warcraft. <http://www.worldofwarcraft.com>.
- [4] World of warcraft system requirements. <http://www.worldofwarcraft.com/info/faq/technology.html>.

- [5] Graham Cormode, S. Muthukrishnan, and Ke Yi. Algorithms for distributed functional monitoring. In *SODA '08: Proceedings of the nineteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 1076–1085, Philadelphia, PA, USA, 2008. Society for Industrial and Applied Mathematics.
- [6] Ankur Jain, Joseph M. Hellerstein, Sylvia Ratnasamy, and David Wetherall. A wakeup call for internet monitoring systems: The case for distributed triggers. In *Proceedings of the 3rd Workshop on Hot Topics in Networks*, 2004.
- [7] M. Ye and L. Cheng. System-performance modeling for massively multiplayer online role-playing games. *IBM Syst. J.*, 45(1):45–58, 2006.
- [8] Ke Yi and Qin Zhang. Multi-dimensional online tracking. In *SODA '09: Proceedings of the Nineteenth Annual ACM -SIAM Symposium on Discrete Algorithms*, pages 1098–1107, Philadelphia, PA, USA, 2009. Society for Industrial and Applied Mathematics.