Pangolin: Speeding up Concurrent Messaging for Cloud-Based Social Gaming

Chao Zhang
Polytechnic Inst. of NYU

Cheng Huang
Microsoft Research

Philip A. Chou
Microsoft Research

Jin Li
Microsoft Research

Sanjeev Mehrotra
Microsoft Research

Keith W. Ross
Polytechnic Inst. of NYU

Hao Chen
Xbox Live

Felix Livni
Xbox Live

Jay Thaler
Xbox Live

Abstract

The convergence of games and online social platforms is an exploding phenomena. The continued success of social games hinges critically on the ability to deliver smooth and highly-interactive experiences to end-users. However, it is extremely challenging to satisfy the stringent performance requirements of online social games.

Motivated by an Xbox Live online social gaming application, we address the problem of *concurrent messaging*, where the maximum latency of game messages has to be tightly bounded. Learning from a large-scale measurement experiment, we conclude that the generic transport protocol TCP, currently being used in the game, cannot ensure concurrent messaging. We develop a new UDP-based transport protocol, named *Pangolin*. The core of Pangolin is an adaptive decision making engine derived from the Markov Decision Process theory. The engine optimally controls the transmission of redundant Forward Error Correction packets to combat data loss. Trace-driven emulation demonstrates that Pangolin reduces the 99.9-percentile latency from more than 4 seconds to about 1 second with negligible overhead.

Pangolin pre-computes all optimal actions and requires only simple table look-up during online operation. Pangolin has been incorporated into the latest Xbox SDK - released in November, 2010 - and is now powering concurrent messaging for hundreds of thousands of Xbox clients.

1. INTRODUCTION

The convergence of games and online social platforms is an exploding phenomena, providing game developers with a new means to rapidly build an enormous user base. The three-year-old social gaming company Zynga now enjoys over 100 million unique players every month to its popular games [2]. Each of the top 25 Facebook games have more than 5 million players every month, and the top 15 Facebook games have more than 10 million monthly players [3]. The rapid growth of social games is propelled by their unique characteristics: 1) social networks enable serendipitous discovery of the games, enabling viral growth without requiring hefty marketing budgets; 2) cloud services make the distribution and update of the games effortless; and 3) an entirely new business model, which gives away the games for free but earns revenue by advertising and selling virtual goods [20].

The continued success of social games, however, hinges critically on the ability to deliver smooth and highly-interactive experiences to end-users. As with many other cloud-based services, the users' perceived performance is greatly affected by latency, delay variation, and packet loss in the Internet. But due to their highly interactive nature, the performance requirements of social games are much more demanding.

Additional challenges arise as many social gaming scenario require *concurrent messaging*, which dictates that every piece of message be delivered (to a specified group of clients) at exactly the same time. We will elaborate using a Xbox Live game in the next section, but one can easily conceive some examples, such as a triggering message to turn signal light from red to green for distributed car racers.

Concurrent messaging is typically implemented in the following way: 1) synchronize all the clients' clock to a virtual clock in the cloud; 2) deliver a message specifying a future virtual clock time stamp; 3) each client processes the message according to the specified virtual clock time stamp and triggers a corresponding event. Clearly, the gap between the current time stamp and the future virtual clock time stamp needs to be large enough, so as to accommodate remote clients with poor network conditions, which might experience timeouts and retransmissions. Hence, the gist of the concurrent messaging problem is to tightly bound the maximum latency for game messages. Unfortunately, as will become clear later, in order to accommodate most clients in the wild, the virtual clock time stamp has to be set to more than 4 seconds into the future. This leads to significant latency inflation and greatly affects the interactive gaming experience.

ACM CoNEXT 2011, December 6–9 2011, Tokyo, Japan.

In this paper, we conduct a large-scale measurement study to gain deep understanding of the concurrent messaging problem. In particular, we design an in-game measurement experiment, develop a measurement engine and release the engine together with the Xbox Live 1vs100 game to hundreds of thousands of Xbox client consoles. Our measurement platform collects detailed packet-level traces from the game clients in the wild. Analysis of these traces shows that the latency, latency variation and packet loss are quite severe for a small but significant percentage of players, which highlights the difficulty of ensuring concurrent messaging, especially for tail clients.

Next, we explore whether the problem can be solved by making tweaks to the existing mechanisms in the generic transport protocol TCP. We develop an emulation platform which replays the collected measurement traces through the TCP stack of real operating system. The experiment allows us to quantitatively evaluate the actual performance of TCP for game clients in the wild. By dissecting the replay results, we also identify a number of factors that make TCP ineffective for concurrent messaging.

It becomes clear that fixing TCP is *insufficient* to solve the problem. To that end, we develop a new transport protocol, named *Pangolin*, which is based on UDP and uses Forward Error Correction (FEC) to speed up concurrent messaging. To keep the overhead low, Pangolin employs an adaptive FEC scheme which dynamically tunes redundancy overhead based on network latency and packet loss rate. While adaptive FEC has been well-explored by many studies (see representative work [9, 14] and their references), we have to address two unique challenges: 1) instead of minimizing latency in most existing schemes, our goal is to minimize the maximum latency for tail clients; 2) optimization techniques in the existing schemes are computationally intensive, which prohibits them from being implemented in already heavily loaded gaming servers.

The core contribution of Pangolin lies in addressing the above challenges by modeling and analyzing the concurrent messaging problem using the Markov Decision Process (MDP) theory. The MDP framework allows us to obtain an optimal adaptive scheme that can tightly bound the maximum message latency, while keeping overhead at minimum. Trace-driven emulation shows that Pangolin reduces the 99.9-percentile latency from more than 4 seconds to about 1 second with negligible overhead. Moreover, optimal actions can be pre-computed and Pangolin simply consults a look-up table of less than 4MB memory during online operation. The small memory footprint and low computation complexity makes it feasible to adopt Pangolin even by gaming servers with already high processing load. As a matter of fact, Pangolin has been incorporated into the latest Xbox SDK - released in November, 2010 - and is now powering concurrent messaging for hundreds of thousands of Xbox clients.

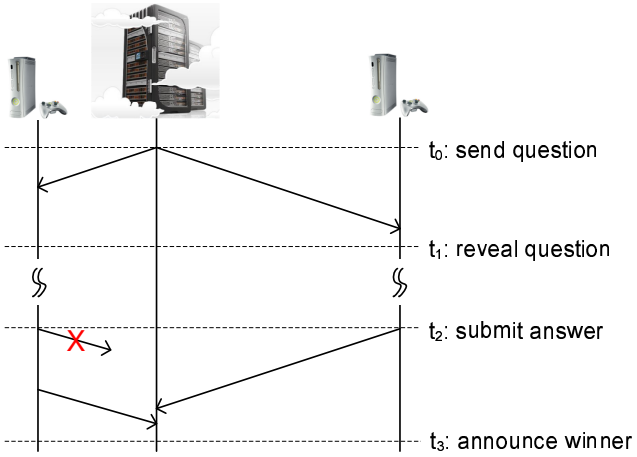


Figure 1: Illustration of Concurrent Messaging (question-reveal delay $= t_1 - t_0$; submit-announce delay $= t_3 - t_2$).

2. MOTIVATING SCENARIO

The Xbox Live *1vs100* game is a massive multiplayer *live social* game [1]. Being a live game, it shares many similarities with live television shows. For instance, each game session occurs only at scheduled time slots (say 8pm on weekends), so all players must participate at the same time. The scale of the game is massive since 1) participants are not divided into small groups, rather there is just one single group for all the participants; 2) the number of concurrent participants is huge, with each session accommodating tens to hundreds of thousands of simultaneous players. Social elements make the game especially appealing, as friends often try to play together to boost their chances of being selected into the Mob [1].

The game session advances in synchronized steps. At the beginning of each step, the game sends a multiple-choice question to all the players at the same time. Due to the network latency, latency variation and packet loss, different players might receive the question at different times.

To ensure that the game is fair to all the players, the question is revealed to all the participants at the same time, based on a single time stamp from a global virtual clock (all the game consoles are synchronized with the host on the same virtual clock). This way, the players closer to the host will not see the question earlier and thus have an unfair advantage. Therefore, at the moment that the question is created, the server needs to determine how far into the future the question should be revealed to the players. Intuitively, the question-reveal delay ($t_1 - t_0$, as shown in Figure 1) should be large enough so that almost all the players receive the question by the reveal time.

After the question is revealed, each player needs to select an answer and submit to the server after a certain deadline. The server must collect nearly all of the answers from the

hundreds of thousands of users before it aggregates all the results and announces the winners. To keep the game exciting and engaging, it is critical that winners be announced shortly after the answer deadline. At the same time, the submit-announce delay ($t_3 - t_2$, as shown in Figure 1) needs to be long enough so that almost all the participants can get their results in.

In both phases (revealing the question and announcing the result), if the delay is too small, due to network latency and loss, then many players will not see the questions on time, or will not be able to get their answers to the server before the deadline, thus making the game unfair. On the other hand, if the latency tolerance is too large, the progress of the game is slowed and the interactivity is severely impaired.

3. MEASUREMENTS AND OBSERVATIONS

3.1 In-Game Measurement Experiment

Detailed packet level traces are very valuable to understand the performance of the existing transport protocol, as well as to assist the design and evaluation of new protocols. One approach to obtain packet level traces is to capture the TCP packets entering and leaving game servers inside the Xbox Live data center. However, traffic dumps cannot reveal one-way packet latency and loss. In addition, accurate inference from traffic dumps can become difficult sometimes [19, 4].

To obtain detailed and truly representative packet level traces, we design an *in-game measurement experiment*. In particular, we have developed and integrated with the 1vs100 game (running in Xbox consoles) a measurement engine. Once activated, the measurement engine replicates real game messages and sends them in UDP packets to a dedicated measurement server in the Xbox Live data center. Note that a Xbox console sends these UDP-encapsulated measurement messages in parallel with the TCP-encapsulated operational messages. Each measurement packet is immediately acknowledged by the measurement server, so that there is no latency inflation due to delayed acknowledgments. Each game message is about 2KB, which is sent in two 1KB UDP packets. To collect more measurement samples, the measurement engine in the console sends three additional UDP packets of the same size, so that a total of five packets per message are sent. Any lost packet (or acknowledgment) is retransmitted after a timeout (up to 5 times). For each transmission, the console records the detailed round trip time and loss information and reports the trace to the measurement server.

The measurement engine has been released together with the 1vs100 game to hundreds of thousands of end-users. The activation of the engine is controlled by the game service, which can turn off the experiment completely. To limit the impact of the measurement experiment to the game itself, only one in every 10 game messages are replicated. The game generates about one message per second, thus the engine generates about one measurement every 10 seconds.

3.2 Trace Collection

During each game session within the two-week period between 2 Feb 2010 and 15 Feb 2010, the measurement engine is activated for a small subset of random clients. Over this period, a total of 10304 end hosts have ever been enrolled in our measurement. Using Quova GeoLocation database, we find that there are 70% unique clients from North America and 30% from Europe, which are the two regions where the game has been released so far. Please refer to [29] for more characteristics about the traces.

3.3 Observation – Delay

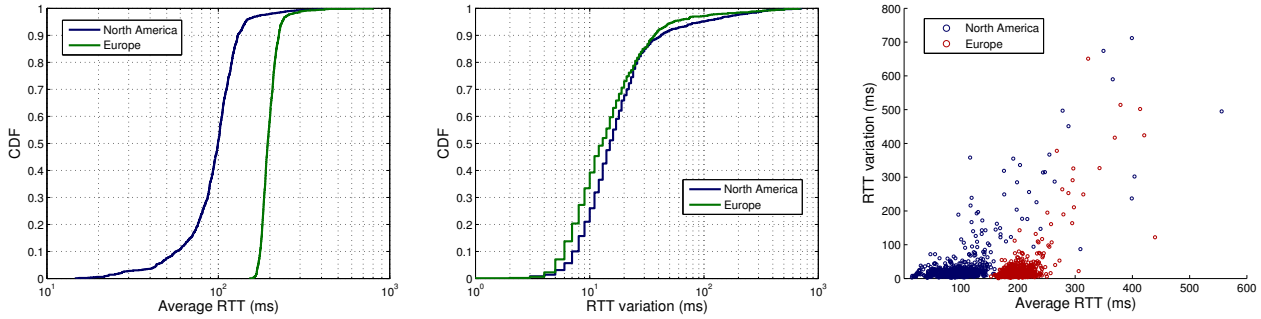
For each client we calculate the average and variation of its RTTs. The distributions of the average RTT for the clients from North America and Europe are plotted in Figure 2(a). Since the game service data center is located in the US, it is not surprising that the average RTT is about 100ms from North America and 200ms from Europe at the 50 percentile. Importantly, at high percentiles, even clients from North America have large RTTs. For example, 2.5% of the clients from North America show an average RTT of more than 200ms, larger than most clients from Europe. This suggests that having the game service data center on the same coninent as the users can only help to a certain extent, since game performance is largely determined by the delays of the high-percentile users.

Next, we examine the RTT variation of each client. As we will later see, RTT variation is an important factor affecting message delivery latency. For each client, we use the difference between the 90-percentile and 10-percentile of its RTT samples to calculate the client's RTT variation. As shown in Figure 2(b), clients from both North America and Europe can experience variations of more than 100ms, which is quite significant. Providing another perspective, Figure 2(c) shows a scatter plot of average RTT and RTT variation. Clearly, large RTT variation occurs predominately with clients with large RTT values. This suggests that the clients with large RTTs, which already have difficulty meeting the basic latency requirements, are also subject to harsh RTT variations.

3.4 Observation – Packet Loss

Now, we examine the packet loss rate experienced by the clients. Figure 3 compares the distributions between North America and Europe. Clearly, a higher fraction of the Europe clients experience loss than North America clients. In addition, the packet loss rate experienced by the clients from Europe is also higher than from North America.

The differences might be due to late-mile connectivity in North America and Europe. Alternatively, it is also possible that packet loss is not solely determined by last-mile, but also affected by middle-miles. Despite of the causes, it is important to note that even in well-developed markets in North America and Europe, where broadband infrastructures are advanced, delay, delay variation and packet loss are still



(a) Average RTT (b) RTT Variation (c) RTT vs. Variation

Figure 2: RTT Distributions.

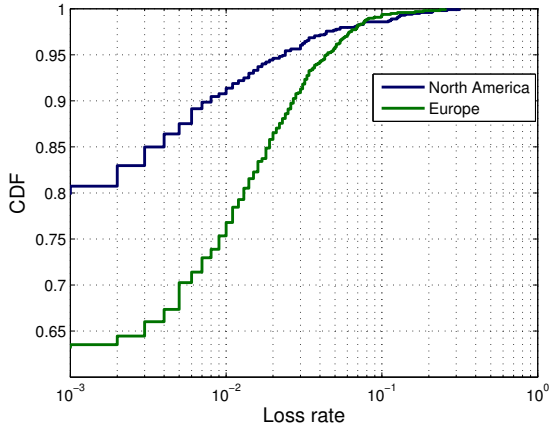


Figure 3: Packet Loss Rate.

severe problems affecting gaming experience.

4. INEFFECTIVENESS OF TCP

4.1 Methodology

In this section, we investigate how TCP performs in the wild for gaming scenarios, for which there are infrequent interactions with small messages between the game consoles and the data center. We explore whether TCP can satisfy the latency requirement of the game and what aspects of TCP may cause poor performance. To answer these questions, one evaluation option is to drive a computer simulation with packet-capture traces, such as done in [5]. However, due to the complexity of TCP implementations, simulation programs unavoidably simplify the workings of TCP and omit important details. To overcome such limitations and evaluate the TCP as it would perform in the real systems, we instead replay our measurement traces through a real TCP stack.

To this end, we borrow the idea of Monarch [18] and develop an emulation platform, which intercepts TCP or UDP packets in kernel and manipulates them in user space based on our measurement traces. Specifically, we create a data sender and a data receiver on the same physical host. Both the sender and the receiver communicate with each other using the regular TCP or UDP stack in the host OS and are *unaware* of the emulation. Nevertheless, packets are captured by a traffic emulator, which manipulates each packet based on a measurement trace: if the trace indicates a loss, then the packet will be dropped; otherwise, the packet will be delayed and forwarded according to the specific latency given in the trace.

4.2 Message Latency with TCP

Using the emulator, we replay all the measurement traces through TCP by sending 2KB messages at one second intervals. We observe that some clients stay longer and contribute more message latency samples than others. To avoid bias towards these clients, we randomly select 10 messages from each client. Table 1 summarizes the quantile results aggregated over all the clients (aggregated separately for North America and Europe). We observe that the message latency is quite small for more than 95% of the messages. However, at high percentiles, the latency becomes very large. Especially, at the 99.9-percentile, which is the performance target of the game, the latency reaches 3 or 4 seconds. In other words, if the game is designed to ensure 99.9% of the messages are delivered on time, the latency tolerance would have to be set to 3 or 4 seconds, resulting in highly degraded interactivity.

	95%	99%	99.5%	99.9%
North America	156	500	765	3284
Europe	264	864	1429	4141

Table 1: Message Delivery Time (ms).

4.3 Analysis

TCP is an all-purpose transport protocol. Being general makes it ineffective in our gaming scenario. We dissect the emulation results to understand each of the factors that inflate message latency and makes TCP unsuitable for many

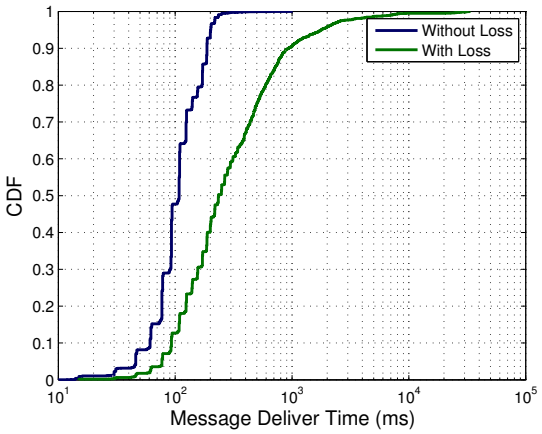


Figure 4: Message Delivery Time (ms).

interactive game services. We study this with an eye on designing a new transport protocol for applications with infrequent short messages with stringent delay requirements. The conclusions are briefly summarized here (please refer to [29] for details).

- TCP keeps only one timer for all outstanding packets. As a result, different loss patterns can incur vastly different message delivery time.
- When the retransmission timer expires, TCP enters into the slow start phase and reduces its congestion window to one. As a result, it takes a very long time to recover two lost packets in the same message.
- Exponential backoff greatly inflates the message delivery time, especially for clients with large RTT.
- The retransmission timeout value (RTO) is calculated quite conservatively.
- Loss is never recovered via *fast-retransmit* due to the small size and infrequency of the game messages. Furthermore, the minimum RTO limits the opportunity to recover loss upon timeout.
- Delayed ACK causes the last packet of many messages not to be acknowledged promptly.
- Due to TCP streaming in-order delivery, the delay of an early message can postpone the delivery of later messages.

The combination of all the above factors collectively causes the occasional (at the 99.9 percentile) large message delivery time, as reported in Table 1. To highlight the impact of these factors, we extract all the messages that are affected by packet loss. We plot the distribution and compare it to those messages not affected by loss at all. As shown in Figure 4, we can observe that more than 10% of the loss-affected messages have final delivery time larger than 1000ms, and packet loss is indeed the dominating factor in inflating the latency.

Fixing all of the above issues calls for a major re-work of the transport protocol. We are therefore compelled to develop an entirely new transport protocol from scratch. The following changes are adopted: (i) there is one timer associated with each message. When the timer expires, all the unacknowledged packets within the message can be retransmitted together [16]. (ii) Exponential backoff and minimum RTO are removed [22, 28]. (iii) ACKs are never delayed beyond message boundaries. (iv) Streaming in-order delivery is no longer a useful semantic and thus removed. (Please refer to [29] for details.)

However, as we will later see, such fixes alone are insufficient to satisfy the game requirement and redundant packet transmission should play a central role in the new protocol. However, the key challenges are: i) how to determine redundancy overhead so as to tightly bound maximum message latency across a wide range of varying network conditions; ii) how to obtain a low complexity solution which is feasible to implement on already heavily loaded gaming servers.

5. PANGOLIN DESIGN

5.1 Rational

The analysis in the earlier sections shows that it is impossible to rely on retransmissions alone to satisfy the delivery requirement, particularly when the latency between client and server is comparable to the latency tolerance. A natural remedy here is to send *redundant* packets together with the data packets so that the redundant packets can be used to recover the original message in the event of packet loss.

To ensure the maximum message latency is tightly bounded, a straightforward solution is to add a fixed number (say r) of redundant packets to each message. If r is chosen aggressively large, the delay requirement can typically be met. However, such a fixed redundancy scheme has a number of problems: i) even when there is no loss, it incurs a fixed overhead of r extra packets per message; ii) even for clients very close to the server, whose latency tolerance is several orders of magnitude larger than the round trip time, it incurs the fixed overhead. For those clients, there is enough time for the clients to wait for timeouts and retransmit lost packets, which obviously is a more economic solution. Therefore, it is also desirable to satisfy the latency requirement with as little overhead as possible.

A dynamic adaptive solution can monitor loss behavior on network links and determine on-the-fly the number of redundant packets to transmit. When clients are close, the solution would simply revert back to pure timeout-based retransmission. When clients are far away, on the other hand, the solution would add just enough redundant packets. For clients in the middle range, the latency tolerance period can be divided into *stages* determined by the round trip time be-

tween the clients and the server. The solution would favor timeout-based retransmissions in the earlier stages and increase the number of redundant packets in the later ones. To summarize, the dynamic adaptive solution can be formulated to solve the following optimization problem – for each individual client, it is desirable to minimize the average number of packets transmitted per message, subject to the constraint that the fraction of messages exceeding the tolerated latency is less than a very small target threshold.

In the rest of the section, we describe an optimal solution to the above problem based on the Markov Decision Process (MDP) theory. Before diving into the solution, it is instructive to first walk through a concrete example and illustrate how the problem is transformed into the MDP framework.

5.2 A Message Transmission Example

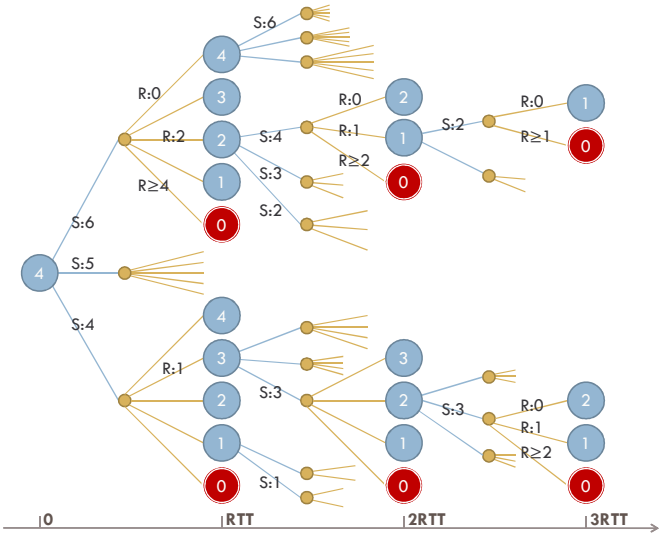


Figure 5: A Message Transmission Example.

Assume a client needs to transmit to the server a message consisting of k data packets. Let T denote the tolerated latency; thus, only messages arriving within T seconds are useful. Further assume there is certain RTT between the client and the server. Suppose that with respect to the RTT, the latency tolerance T can be divided into 3 stages, as shown in Figure 5. At the beginning of stage i ($i = 1$, 2 or 3 here), there are q data packets to be transmitted. We define the *state* as (i, q). For instance, if the original message consists of $k = 4$ packets, then there are 4 packets to be transmitted at the beginning of stage 1, and the initial state is $(1, 4)$. Given a state (i, q), we need to choose a transmission *action*, which is the total number of packets to transmit and is denoted by $\pi(i, q)$. One exemplary action shown in Figure 5 is $\pi(1, 4) = 6$, which is to transmit 6 packets at stage 1, marked as "S:6". This means that 6 packets will be transmitted in stage 1, including 4 data packets and 2 *redundant* packets.

Continuing with this example, when packet loss occurs, the server will receive less than 6 packets. If it receives 4 or more packets, the server can recover the original message. In such cases, the message is successfully delivered in stage 1 and there is no need to transmit more packets in the next stages. This is marked in Figure 5 as a special ending state "0" at the end of stage 1. Otherwise, assuming the server only receives 2 packets (no matter data or redundant packets, marked as "R:2" in Figure 5), then there are still 2 more data packets that need to be transmitted in the second stage. Therefore, the current state becomes $(2, 2)$. Again, another transmission action is to be chosen for the new state. Depending on the loss events in the network, the action will lead to another state at the end of stage 2, and so on. By the end of stage 3, the message still may not be delivered completely, i.e., the state at the end of stage 3 is non-zero. In this case, this message fails to satisfy the latency tolerance.

5.3 Problem Formulation in MDP Framework

The choice of the action at each given state, combined with the probabilistic nature of packet loss, creates a stochastic process. As illustrated by the trellis in Figure 5, the action and packet loss determines the evolution of the process. In the terms of the Markov Decision Process theory, a mapping from states to actions is called a *policy*, denoted as $\pi = \{\pi(i, q)\}$. For a given policy π, the action at each state (i, q) is deterministic (defined by the policy itself). In illustrations similar to Figure 5, each deterministic action corresponds to a single link leaving a state. Packet losses are, however, probabilistic and different loss patterns result in different states, corresponding to different branches ending up at different states.

For a given message, let p denote the current estimate of the packet loss probability. Also, let I denote the number of stages. The probability of each path can be calculated as the compound probability of all the loss patterns along the path. Aggregating all the paths ending at non-zero states in the final stage I (or stage 3 in the above example), we can obtain the probability that the message cannot be delivered by T. This probability is equivalent to the fraction of messages arriving after T, which we denote as ϵ_π. Here, the subscript π represents the associated policy. The cost of each path can similarly be calculated as the total number of packets transmitted along the path. Aggregating the costs on all the paths weighted by their probabilities, we can obtain the average cost of delivery a message, which we denote as ρ_π.

To ensure that the majority of messages are delivered within the latency tolerance, we require the percentage of the messages arriving later than T be less than a very small target threshold, denoted as $\epsilon(T)$.

Therefore, the original optimization problem can be transformed into the MDP framework to find the optimal policy, which minimizes the average transmission cost while at the same time ensuring the probability of the message arriving later than T is below the threshold. In particular, the optimization problem becomes

$$\min_{\pi} \rho_{\pi}$$
$$\text{s.}t.\ \epsilon_{\pi} \leq \epsilon(T).$$

5.4 An MDP-Based Solution

The above constrained MDP problem can be converted to an unconstrained MDP problem using standard Lagrangian methods. To this end, we first introduce a Lagrangian multiplier λ and define a combined objective function as a weighted sum of the failure probability and transmission cost, denoted by

$$J_{\pi}(i,q) = \epsilon_{\pi}(i,q) + \lambda\rho_{\pi}(i,q). \quad (1)$$

Thus $J_{\pi}(i,q)$ is the combined cost when beginning in the sub-trellis rooted in state (i,q). Obviously, the objective function over the entire trellis is $J_{\pi}(1,k)$. Next, for a given λ, we solve the modified optimization problem which minimizes $J_{\pi}(1,k)$ and finds the optimal policy π^*, i.e.,

$$\pi^* = \arg\min_{\pi} J_{\pi}(1,k). \quad (2)$$

For the optimal policy π^* determined by a given λ, the message failure rate $\epsilon_{\pi}(1,k)$ can be readily evaluated. It might or might not satisfy the latency requirement constraint ($\epsilon_{\pi} \leq \epsilon(T)$). Hence, the final step is to vary λ and find the closest value through bi-section search that just satisfies the constraint. This is equivalent to finding a point on the convex-hull along the trade-off curve between the message failure rate and the transmission cost [8, 14].

To solve the above modified optimization problem, the objective function of a particular trellis can be expressed in terms of its sub-trellises, as

$$J_{\pi}(i,q) = \lambda\pi(i,q) + \sum_{q'=0}^{q} p(q'|q,\pi(i,q))J_{\pi}(i+1,q'), \quad (3)$$

where $p(q'|q,\pi(i,q))$ represents the transitional probability from state (i,q) to state $(i+1,q')$ by transmitting $\pi(i,q)$ number of packets. Given the model of packet loss, the transition probability can be readily calculated. For example, assuming the packet loss rate is uniform and denoted by p, the transition probability is calculated as Equation 4. Also, the cost at the edge is computed as $J_{\pi}(I+1, q \neq 0) = \epsilon_{\pi}(I+1,q) + \lambda\rho_{\pi}(I+1,q) = 1$, since the failure probability and the transmission cost after the final stage I are $\epsilon_{\pi}(I+1,q) = 1$ and $\rho_{\pi}(I+1,q) = 0$, respectively. Of course, $J_{\pi}(I+1, q=0) = 0$.

Let $J^*(i,q)$ and $\pi^*(i,q)$ define the minimum value of the objective function and the corresponding action, over the sub-trellis rooted at (i,q). Then

$$J^*(i,q) = \min_{a} \big(\lambda a + \sum_{q'=0}^{q} p(q'|q,a)J^*(i+1,q')\big), \quad (5)$$

$$\pi^*(i,q)) = \arg\min_{a} \big(\lambda a + \sum_{q'=0}^{q} p(q'|q,a)J^*(i+1,q')\big). \quad (6)$$

By induction, it can be readily shown that $J^*(i,q) \leq J_{\pi}(i,q)$ for all (i,q) and all π, with equality achieved when $\pi = \pi^*$.

Therefore, the problem of finding the optimal policy π^* (i.e., Equation 2) can be solved efficiently using *dynamic programming* with the recursive Equation 5 and Equation 6. We briefly note here that solution procedure just outlined above will not find the optimal policy in exact, since the constraint will typically not be met in equality. To achieve optimality, we need to introduce some randomization into the selection of actions [8]. However, the deterministic policy derived above will be nearly optimal and sufficient for practical purposes.

5.5 Computation Complexity

We remark that the optimal policy is completely determined given the packet loss probability, the initial state (i.e., the number of data packets in a message), as well as the ratio between RTT and the latency tolerance (i.e., the number of stages I). With these inputs, the dynamic programming problem outlined above can be solved for any Lagrangian multiplier λ. Then, through bi-section search, the optimal policy can be readily found [8, 14].

Based on this observation, to avoid performing expensive optimization computations online, in practice, all the optimal policies are pre-calculated and stored as look-up tables for all possible combinations of (quantized) inputs. During online adaptation, the optimal action can be obtained with a simple table lookup, given the current state (how many data packets remain to be transmitted), the stage of transmission, the packet loss rate, the RTT and the latency tolerance.

It might appear that a large number of policy tables would have to be stored and thus consume significant amount of memory. In reality, many tables are *trivial* and thus don't need to be stored – when packet loss rate is relatively low and/or the number of stages is relatively large, the optimal action at every single stage is simply to transmit all the remaining data packets *without* any parity packet. These optimal action tables, once pre-computed, do *not* need to be stored physically. Therefore, the actual memory consumption can be significantly reduced. Assume the packet loss probability varies between 0 and 50%. With a quantization step of 1%, there are $P = 50$ different packet loss probabilities. Also, assume the latency tolerance is 1000ms. With a RTT quantization step of 10ms, there are up to $R = 1000/10 = 100$ transmission stages. Finally, each message can have up to $K = 64$ packets. Instead of $P \times R \times K = 320K$ tables, in the end, we only have to store 22,968 tables

$$p(q'|q,a) = \begin{cases} 0 & \text{if } q < q' \text{ or } q > q' + a \\ \sum_{r=q}^{a} \binom{a}{r}(1-p)^r p^{(a-r)} & \text{if } q' = 0 \\ \binom{a}{q-q'}(1-p)^{(q-q')}p^{(a-(q-q'))} & \text{if } 0 < q' \leq q \leq q' + a \end{cases} \quad (4)$$

with less than 4MB in total size. Clearly, all the tables can be pre-loaded by both game consoles and data center servers easily.

5.6 Network Congestion

The rate of concurrent messaging in Xbox games is typically quite low, such as one message per second. Network congestion is less of a concern at such low rate. However, if Pangolin were to be used to deliver large messages at high rates, the adaptive scheme needs to properly incorporate network congestion signal. The Pangolin scheme has been extended to cope with network congestion and we refer interested readers to [29] for the details.

6. ARCHITECTURE AND IMPLEMENTATION

Based on the above adaptive FEC scheme, we have implemented the Pangolin transport protocol to ensure that message latency requirement is satisfied without incurring high transmission cost. Pangolin is a message-oriented, connectionless, UDP-based protocol. In this section, we describe its architecture and implementation. (Please refer to [29] for protocol details.)

The Pangolin protocol is built on top of UDP. It includes a Pangolin core and a set of APIs. The Pangolin core implements all the major functions of a full-fledged transport protocol, such as 1) maintaining per-flow status for each communicating end-point; 2) estimating the parameters of communication channels, such as round trip time, packet loss rate, timeout period, etc.; 3) delivering packets using a combination of FEC and retransmission, as determined by the adaptive algorithm; and so on. The Pangolin APIs support both synchronous and asynchronous message transfers.

The core of the Pangolin stack contains about 7000 lines native C++ and about 500 managed C++ code to support C# programming[1]. The majority of the codebase lies in the implementation of the UDP-based transport protocol stack and performance optimizations for the stack. The MDP-based adaptive solution only accounts for a tiny fraction of the codebase because it is as simple as – whenever a message (re)transmission event arises, performing a table-lookup based on the number of unacknowledged packets (in the message) and transmitting specified number of packets based on the table entry. It is also worth noting that all the performance optimizations for the protocol stack, such as thread pool, completion port for efficient network I/Os, etc., are applied only on server side. The protocol stack went into Xbox consoles is significantly smaller.

[1]Xbox consoles use the native APIs, while game servers in the data center use the C# APIs.

7. PERFORMANCE EVALUATION

In this section, Pangolin is first evaluated under simple exemplary scenarios, such as fixed round trip time and constant packet loss rates. Focusing on such simple scenarios allows us to fully understand the trade-off between the transmission cost and the probability of satisfying the latency requirement, as well as the advantages of Pangolin over the TCP and fixed FEC schemes. Pangolin is then evaluated using the real-world packet-level trace, which further confirms its benefit for end-users in the wild.

7.1 Transmission Cost

We start with simple exemplary scenarios. Assume the round trip time between a client and the data center is 250ms and the target latency is 500ms. Therefore, the client has $500/200 = 2$ rounds of transmission opportunity. Further assume the message is divided into $k = 4$ data packets and the packet loss rate is $p = 2\%$. Thus, the optimal transmission policy derived from the MDP solution is $\pi^*(q_1 = k_1) = k_1$ and $\pi^*(q_2 = k_2) = k_2 + 1$, i.e., in the first round, $k = 4$ packets are transmitted; and in the second round, $k' + 1$ packets are transmitted, given there are k' remaining *data* packets. The expected transmission cost can also be calculated and is $\rho^* = 4.16$. Then, a *normalized transmission overhead* (simply *overhead* hereafter) can be calculate as $(\rho^* - k)/k$, which is $(4.16-4)/4 = 4\%$ and is very small.

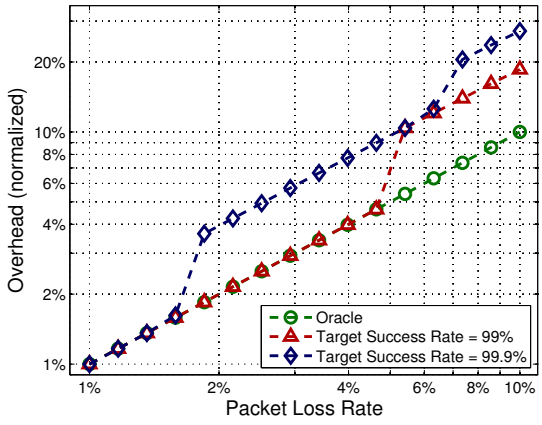


Figure 6: Transmission Cost of Optimal Policies.

Figure 6 shows the overhead of optimal policies under a wide range of packet loss rates. We observe that ensuring 99.9%, rather than 99%, of the messages delivered within the latency constraint is much more difficult, as the corresponding overhead is noticeably higher. In addition, for a given target success rate, interestingly, the overhead curve appears to be piece-wise linear. For example, the curve corresponding to the target success rate of 99.9% consists of three linear segments. Further investigation reveals that there are only *three* different policies across the entire range of packet loss rates, and each linear segment is covered by the same policy. This implies that the optimal policies are in fact *not* sensitive to packet loss rate. Therefore, the very estimation of packet loss rate does *not* have to be highly accurate. Finally, the minimum overhead curve is also plotted as a comparison, which is calculated using an *oracle* scheme that knows all the losses *a priori* and adds FEC accordingly. We can conclude that, even targeting a high success rate of 99.9%, the actual overhead is not much higher than the oracle scheme. This further confirms the effectiveness of the optimal policies.

7.2 Message Latency

In this section, we evaluate the performance of the real Pangolin stack by emulating similar network conditions. Specifically, the emulated round trip time is 250ms and the packet loss rate 2% (on the forward connection only). The message size is 2KB and divided into 4 packets. The target latency is 500ms, or twice the round trip time.

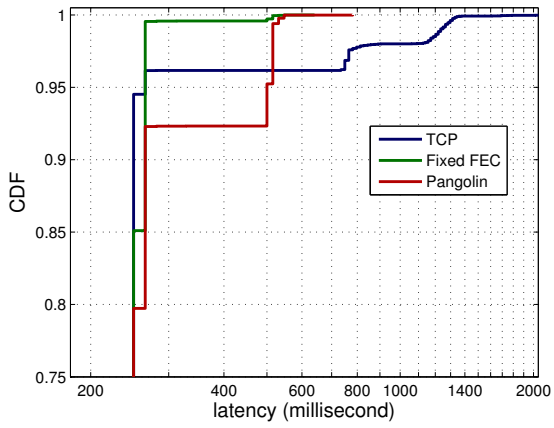


Figure 7: Latency Performance Comparison.

For comparison purposes, a fixed FEC scheme is also evaluated. This scheme always transmits one FEC packet together with the four data packets and hence has a constant overhead of 25%. Figure 7 compares the message latency performance among TCP, the fixed FEC scheme and Pangolin. The results at high percentiles are also summarized in Table 2.

From the comparison, we can observe that no matter which protocol is used, a large percentage of the messages are delivered at the minimum latency ∼250ms (the small deviation is introduced by the network emulator). This is intuitive since the probability of packet loss is low. However, at high percentiles, the latency of TCP quickly increases and reaches more than 1.3 seconds at 99.9%, which is very significant compared to the round trip time of 250ms. On the other hand, both the fixed FEC scheme and Pangolin are able to satisfy the 500ms latency requirement. In addition, we observe that the fixed FEC scheme delivers most messages at the minimum latency, while Pangolin delivers more than 7% of the messages at the target latency. Clearly, the fixed FEC scheme is over-aggressive and the adaptive scheme of Pangolin is sufficient. This also explains why Pangolin only requires 4.3% overhead, much less than the fixed FEC scheme. Finally, we comment that the 4.3% overhead as measured in the Pangolin stack matches nicely with the theoretical value from Section 7.1.

Latency (ms)	TCP	Fixed FEC	Pangolin
95%	266	266	500
99%	1234	266	516
99.9%	1359	516	531
Overhead	-	25%	4.3%

Table 2: Latency Performance Comparison.

7.3 Trace-Driven Emulation

7.3.1 Latency

Latency (ms)	TCP	Retrans.	Fixed FEC	Pangolin
95%	219	219	204	219
99%	566	469	328	412
99.5%	828	703	469	581
99.9%	2352	1501	984	938
Overhead	-	3.9%	150%	6.1%

Table 3: Latency Performance Comparison in the Wild.

In this section, we evaluate the performance of Pangolin in the wild. We use Pangolin to send 2KB messages and control individual packets by replaying the collected measurement traces. We compare Pangolin to TCP, a retransmission-only scheme, and a fixed FEC scheme which always uses the same number of redundant packets. In Pangolin, each 2KB message is split into $k = 2$ packets. In the experiment, the latency tolerance T is set to be 1000ms.[2] The target success rate is set at 99.9%, i.e., 99.9% of the messages should be delivered within T. For the fixed FEC scheme, 3 redundant

[2]We remove all the clients whose average RTT is greater than 1000ms, as they cannot satisfy the latency requirement even in the absence of packet loss. There are 19 such clients.

packets are added, i.e. 5 packets are sent in total[3].

Table 4 shows the comparison results. The most important performance metric is the latency at the 99.9-percentile. We observe that the retransmission-only scheme, which fixes all the issues of TCP as discussed in Section 4, reduces the latency from 2.3 seconds to 1.5 seconds. However, it is also clear that fixing TCP alone cannot solve the problem, as the latency still does *not* satisfy the requirement. In comparison, the fixed FEC scheme can achieve the desirable latency, but at very high overhead (at least 150% in this case). In contrast, Pangolin is not only able to satisfy the latency requirement, but also uses much less overhead than the fixed FEC scheme. Furthermore, Figure 8 shows the CDF of the four schemes, where Figure 8(b) is a zoomed view at high percentile. From the figures, we can see that Pangolin shows higher message latency than the fixed FEC scheme most of the time. This is due to the design of Pangolin's adaptive scheme, which does *not* seek to minimize delivery time, but rather to meet the target latency with minimum overhead. Finally, at the target percentile, Pangolin is able to satisfy the required latency while the retransmission-only scheme cannot.

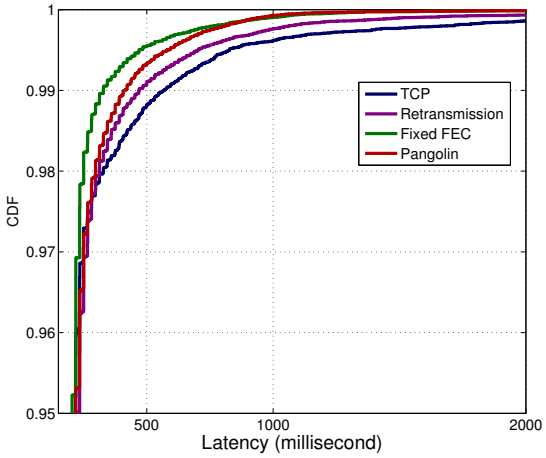
7.3.2 Cost

Latency (ms)	Retrans.	k=2	k=3	k=4	k=5
95%	219	219	234	234	235
99%	469	412	438	457	469
99.5%	703	581	610	623	631
99.9%	1501	938	985	1000	1015
Overhead	3.9%	6.1%	7.9%	9.8%	12.0%

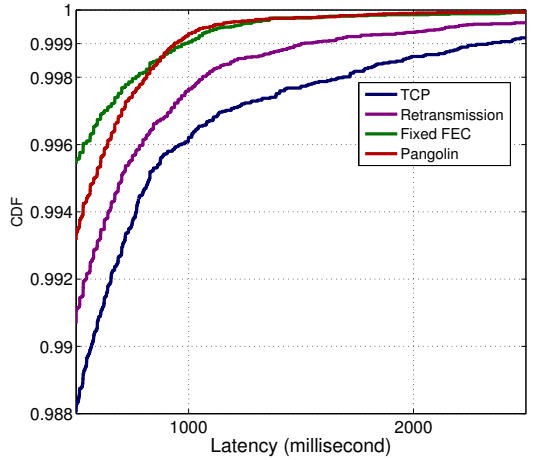
Table 4: Latency and Cost Comparison in the Wild.

Next, we evaluate the overhead introduced by Pangolin and its relationship with packet size. Each 2KB message is divided into k data packets. Intuitively, when k is small, each FEC packet, of the same size as the data packet, is larger and thus more costly. On the other hand, when k is large, the same level of redundancy requires more FEC packets and thus more protocol overhead. Therefore, it is not clear what is the best choice of k. Here, we conduct a simple empirical evaluation by comparing all reasonable k values, from 2 to 5. The protocol header of Pangolin is 12 bytes, so for every packet, the total protocol overhead is 40 bytes, including IP and UDP headers. The overhead of Pangolin is calculated as the ratio between the total number of bytes transmitted (including all protocol headers) and the total number of bytes of all the messages (without protocol header). The results are shown in Table 4. Interestingly, we observe that as the k increases, the overhead of Pangolin also increases. Also, $k = 2$ yields the minimum overhead, which is not much higher than the minimum overhead of the retransmission-only scheme.

[3]One or two redundant packets are also tried, but neither satisfies the latency requirement. Hence, the fixed FEC scheme works only in retrofitting and is hardly practical.



(a)



(b)

Figure 8: Latency Performance Comparison in the Wild.

8. RELATED WORK

8.1 Replacing TCP

Being a generic transport protocol, TCP cannot be optimized for every specific scenario. Based on unique application characteristics and requirements, there have been many efforts to replace TCP with customized UDP-based transport protocols. The most significant ones are perhaps [25, 27]. In a data center environment, when hundreds of machines start TCP transfers in parallel, the so-called *network incast* problem can happen as the sudden burst incurs significant packet loss. The loss in turn leads to a large silent period because of large TCP timeout value, which greatly inflates transfer latency. To address this problem, Facebook built a

custom congestion-aware UDP-based transport protocol to manage congestion across multiple flows and avoid the default 200ms TCP timeout [25]. In peer-to-peer data sharing, in order to be friendly to interactive applications and tame the queuing delay in bottlenecks, BitTorrent has developed a custom transport protocol and moved significant amount of peer-to-peer traffic over the Internet to UDP. The new protocol allows BitTorrent to deploy a novel congestion control algorithm, which can fully utilize the available capacity at the presence of competing TCP flows and yield into background when their presence is detected [27]. Pangolin is similar in its design philosophy. We recognize that a major re-work of TCP is warranted, but that alone cannot solve the problem. Hence, Pangolin is designed as a new UDP-based protocol. Of course, Pangolin targets the concurrent messaging problem, which is completely different from the above settings and thus bears a totally different solution.

8.2 FEC as an End-to-end Technology

Besides being widely used in physical-layer communications, Forward Error Correction codes are also explored as an end-to-end technology to protect data and combat application-layer packet loss. Based on the level of protection requirement, the applications using FEC can be broadly classified into three categories.

8.2.1 Real-time Communications

For this application, data loss is tolerable, as it only degrades perceived performance, but doesn't cause complete disruption. In interactive video conferencing, Rhee et al. [24] proposed to recover from error propagation using continuous updates, which allow FEC packets to be transmitted even after the playback of their associated frames. For VoIP traffic, Bolot et al. [10] used measurements over the Internet and showed that most loss periods involve only a small number of packets. Therefore, an open loop FEC-based scheme was adequate to significantly improve quality. For live streaming broadcast, Chou et al. [14] developed a combined FEC and pseudo-ARQ scheme, which splits video signal into layers and each layer into FEC-coded sub channels. A receiver subscribes to the layers and channels that optimize quality based on its bandwidth and packet loss rate. Similarly, Zattoo [13] uses FEC substreams to protect data substreams in peer-to-peer live streaming.

8.2.2 Bulk Data Transfer

In this application, large data objects of non-degradable nature are to be transferred. Data loss is intolerable and the objects have to be received in their entirety. High transfer throughput is the first order objective and FEC codes provide a very efficient way to deal with packet loss when broadcasting to a large number of receivers [23, 26] or retrieving from multiple sources [12]. Efficient codes are also developed to reduce computation complexity, such as digital fountain and online codes [11, 21]. Finally, network coding approach based on random linear codes also provides an alternative to improve bulk transfer throughput [15, 17].

8.2.3 Short Transfers

These applications require transferring non-degradable short messages within very short latencies. Balakrishnan et al. [6, 7] developed schemes to deal with such transfers within data centers and across data centers. Pangolin belongs to this type of application. Different from the above work, where the primary focuses are the design of special FEC codes based on the characteristics of the applications, Pangolin uses standard FEC codes. The focus of Pangolin is an adaptive decision scheme to minimize overhead while satisfying the latency tolerance requirement.

8.3 Adaptive Schemes

Since FEC incurs extra overhead, it is clear that a redundancy scheme involving FEC should be adaptive. When there is no packet loss, no redundancy should be added. On the other hand, when packet loss rate is high, the level of redundancy should be high. Bolot et al. [9] studied the adaption problem in the context of VoIP. In particular, they formulated a constraint optimization problem and answered the following questions: 1) when should FEC packets be sent? and 2) what source rate should each packet include? Pangolin differs from their work in two important ways: 1) instead of assuming an open loop control, Pangolin relies heavily on feedback. Pangolin uses retransmission whenever possible, which is why it can significantly reduce overhead; 2) instead of solving complex optimization problems online, Pangolin pre-computes all the optimal policies and the online adaptation only requires simple table lookup. This makes it possible for Pangolin to be adopted even by gaming servers with high processing load.

The work of Chou et al. [14] developed a rate-distortion optimized framework to deliver packetized media. The RaDiO framework schedules which packets to send in order to meet a deadline constraint while minimizing the end-to-end distortion. The adaptive scheme in Pangolin is largely inspired by this work. However, the important differences are: 1) instead of relying on ARQ and retransmission, Pangolin combines retransmission with FEC codes and unifies the scheduling of both types of packet; 2) Pangolin focuses on minimizing overhead, which is a totally different performance metric from the distortion.

9. CONCLUSION

In this paper, we address the problem of concurrent messaging for cloud-based social gaming. Learning from a large-scale measurement experiment, we conclude that the generic transport protocol TCP, currently being used in Xbox Live online games, cannot provide concurrent messaging to the game players. We develop Pangolin, a new UDP-based transport protocol, which uses an adaptive decision making engine to combat loss with redundant FEC packets. Both theo-

retical analysis and trace-driven emulation demonstrate that Pangolin minimizes the 99.9-percentile while keeping the overhead negligible. Pangolin has been incorporated into the latest Xbox SDK – released in November, 2010 – and is now powering concurrent messaging for hundreds of thousands of Xbox clients.

10. REFERENCES

[1] 1 vs. 100 (Xbox 360). In *Wikipedia*.

[2] Zynga – the world's most intriguing startups. In *Businessweek* (November 2009).

[3] The 25 largest facebook games as 2010 begins. In *Inside Facebook* (January 2010).

[4] ALLMAN, M., EDDY, W. M., AND OSTERMANN, S. Estimating loss rates with tcp. In *ACM SIGMETRICS Performance Evaluation Review* (2003).

[5] ALLMAN, M., AND PAXSON, V. On estimating end-to-end network path properties. In *SIGCOMM '99: Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication* (New York, NY, USA, 1999), ACM, pp. 263–274.

[6] BALAKRISHNAN, M., BIRMAN, K., PHANISHAYEE, A., AND PLEISCH, S. Ricochet: Lateral error correction for time-critical multicast. In *Fourth Usenix Symposium on Networked Systems Design and Implementation* (2007).

[7] BALAKRISHNAN, M., MARIAN, T., BIRMAN, K., WEATHERSPOON, H., AND VOLLSET, E. Maelstrom: Transparent error correction for lambda networks. In *Fifth Usenix Symposium on Networked Systems Design and Implementation*.

[8] BEUTLER, F. J., AND ROSS, K. W. Optimal policies for controlled markov chains with a constraint. In *Journal of Mathematical Analysis and Application* (1985).

[9] BOLOT, J.-C., FOSSE-PARISIS, S., AND TOWSLEY, D. Adaptive fec-based error control for internet telephony. In *IEEE INFOCOM* (1999).

[10] BOLOT, J. C., AND VEGA-GARCIA, A. The case for fec-based error control for packet audio in the internet. In *ACM Multimedia Systems* (1997).

[11] BYERS, J., LUBY, M., MITZENMACHER, M., AND REGE, A. A digital fountain approach to reliable distribution of bulk data. In *Proceedings of ACM SIGCOMM* (1998).

[12] BYERS, J. W., CONSIDINE, J., MITZENMACHER, M., AND ROST, S. Informed content delivery across adaptive overlay networks. In *Proceedings of ACM SIGCOMM* (2002).

[13] CHANG, H., JAMIN, S., AND WANG, W. Live streaming performance of the zattoo network. In *ACM Internet Measurement Conference* (2009).

[14] CHOU, P. A., AND MIAO, Z. Rate-distortion optimized streaming of packetized media. In *IEEE Transactions on Multimedia* (Apr. 2006).

[15] CHOU, P. A., WU, Y., , AND JAIN, K. Practical network coding. In *Allerton Conference on Communication, Control, and Computing* (2003).

[16] DUKKIPATI, N., REFICE, T., CHENG, Y., CHU, J., SUTIN, N., AGARWAL, A., HERBERT, T., AND JAIN, A. An argument for increasing tcp's initial congestion window. In *ACM Sigcomm CCR* (July 2010).

[17] GKANTSIDIS, C., AND RODRIGUEZ, P. Network coding for large scale content distribution. In *IEEE INFOCOM* (2005).

[18] HAEBERLEN, A., DISCHINGER, M., GUMMADI, K. P., AND SAROIU, S. Monarch: a tool to emulate transport protocol flowsover the internet at large. In *IMC '06: Proceedings of the 6th ACM SIGCOMM conference on Internet measurement* (New York, NY, USA, 2006), ACM, pp. 105–118.

[19] JIANG, H., AND DOVROLIS, C. Passive estimation of tcp round-trip times. In *ACM CCR* (2002).

[20] LIEW, J. Why the economics of social gaming are so attractive to investors, December 2009.

[21] MAYMOUNKOV, P., AND MAZIRES, D. Rateless codes and big downloads. In *The Second International Workshop on Peer-to-peer Systems* (2003).

[22] MONDAL, A., AND KUZMANOVIC, A. Removing exponential backoff from tcp. In *ACM SIGCOMM Computer Communication Review* (2008).

[23] NONNENMACHER, J., BIERSACK, E., AND TOWSLEY, D. Parity-based loss recovery for reliable multicast transmission, 1998.

[24] RHEE, I., AND JOSHI, S. R. Fec-based loss recovery for interactive video transmission experimental study. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems* (1998).

[25] ROTHSCHILD, J. High performance at massive scale: Lessons learned at facebook. In *CNS Lecture Series* (2009).

[26] RUBENSTEIN, D., KASERA, S., TOWSLEY, D., AND KUROSE, J. Improving reliable multicast using active parity encoding services (apes). In *Computer Networks* (2004).

[27] SHALUNOV, S. Low extra delay background transport (ledbat). In *IETF Draft* (2009).

[28] VASUDEVAN, V., PHANISHAYEE, A., SHAH, H., KREVAT, E., ANDERSEN, D., GANGER, G., GIBSON, G., AND MUELLER, B. Safe and effective fine-grained tcp retransmissions for datacenter communication. In *ACM SIGCOMM* (2009).

[29] ZHANG, C., HUANG, C., CHOU, P. A., LI, J., MEHROTRA, S., ROSS, K. W., CHEN, H., LIVNI, F., AND THALER, J. Pangolin: Speeding up Concurrent Messaging for Cloud-Based Social Gaming. In *Technical Report, Microsoft Research* (June 2011). `http://research.microsoft.com/~chengh/techreports/pangolintech.pdf`.