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 the 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 game messages are required to deliver to game players in the wild at exactly the same time. 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 maximum latency (for nearly all users) from more than 4 seconds to about 1 second, while keeping the overhead negligible.

Pangolin pre-computes all the optimal decisions and requires only simple table look-up during online operations. Pangolin has already been incorporated into the latest Xbox SDK, which was 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 [21].

The continued success of social games, however, critically hinges on the ability to deliver smooth and highly-interactive experiences to the end-users. As with many other cloud-based services, the user’s 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 also arise as many social gaming scenario require concurrent messaging, which dictates that every piece of message being 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. As will become clear later, in order to accommo-
date 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.

In this paper, we start with a large-scale measurement study to gain a deep understanding of the concurrent messaging problem. 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. Our 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 a 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 not sufficient to solve the problem. To that end, we develop a new transport protocol, named Pangolin, that 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 the FEC redundancy based on network latency and packet loss rate. Although 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 greatly reduce overhead while ensuring concurrent messaging for nearly all clients. Our trace-driven emulation shows that Pangolin reduces the maximum latency (for nearly all clients) from more than 4 seconds to about 1 second while keeping the overhead negligible. Moreover, the optimal actions can be pre-computed and Pangolin simply consults a look-up table of less than 4MB memory during online operations. The small memory footprint and low computation complexity makes it possible for Pangolin to be adopted even by gaming servers with high processing load. As a matter of fact, Pangolin has already been incorporated into the latest Xbox SDK, which was released in November, 2010 and is now powering concurrent messaging for hundreds of thousands of Xbox clients.

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 the game server 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 \[20, 4\].

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 consoles record the detailed round trip time and loss information and report their traces 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. The distributions of how long each client stays in the game (or time-in-game) is shown in Figure 2. In our log, about 5% players stay in the game over 15 minutes, and only 1% users stay over 40 minutes.

3.3 Observation – Delay

For each client we calculate the average and variation of their RTTs. The distributions of the average RTT for the clients from North America and Europe are plotted.
in Figure 3(a). Since the game service data center is located in the US, it is not surprising that the average RTT is about 100ms from NA and 200ms from EU at the 50 percentile. Importantly, at high percentiles, even clients from NA have large RTTs. For example, 2.5% of the clients from NA show an average RTT of more than 200ms, larger than most clients from EU. This suggests that having the game service data center on the same continent 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 3(b), clients from both NA and EU can experience variations of more than 100ms, which is quite significant. Providing another perspective, Figure 3(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 4 compares the distributions between NA and EU. Clearly, a higher fraction of the EU clients experience loss than NA clients. In addition, the packet loss rate experienced by the clients from EU is also higher than from NA. It therefore appears that packet loss is not solely determined by the last mile but also that the middle-miles can non-negligibly contribute.

4 Understanding TCP Ineffectiveness

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 console 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, as shown in Figure 5, we create a data sender and a data receiver on the same physical host. Both the sender and

Figure 3: RTT Distributions.

Figure 4: Packet Loss Rate.
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 NA and EU). 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.

<table>
<thead>
<tr>
<th></th>
<th>95%</th>
<th>99%</th>
<th>99.5%</th>
<th>99.9%</th>
</tr>
</thead>
<tbody>
<tr>
<td>US</td>
<td>156</td>
<td>500</td>
<td>765</td>
<td>3284</td>
</tr>
<tr>
<td>EU</td>
<td>264</td>
<td>864</td>
<td>1429</td>
<td>4141</td>
</tr>
</tbody>
</table>

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. In this section, we dissect the emulation results to understand each of the factors that inflate message latency and make TCP unsuitable for many 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.

4.3.1 Loss pattern

We first analyze TCP behavior when various packet loss patterns are encountered. A 2KB message is transmitted in 2 TCP packets. Obviously, there are 4 different loss patterns: no loss; only the first packet is lost; only the second one is lost; or both packets are lost. Due to TCP’s internal mechanisms of handling loss, these patterns can lead to vastly different message latencies.

Clearly, without packet loss, the message delivery time is determined by the the packet with the largest RTT, that is \(\max(RTT_1, RTT_2)\). When a single packet is lost, however, depending on whether it is the first or the second packet, the message delivery time differs greatly. This is due to the fact that TCP keeps only one timer for all outstanding packets. When both packets are transmitted back-to-back, only the first one is timed. In the case that the first one is lost, the packet is retransmitted after the retransmission timer expires. The delivery time thus is \(\max(RTO, RTT_2) + RTT_1\), where \(RTO\) is the retransmission timeout. On the other hand, if the second one is lost, the packet will not be retransmitted until \(RTT_1 + RTO\). This is because the retransmission timer for this packet is not started until the acknowledgment for the first packet is received. Hence, the message delivery time becomes \(RTT_1 + RTO + RTT_2\). Figure 6(a) and 6(b) plot these two cases. The single timer mechanism used in TCP increases delivery time.

Now we analyze what happens when both packets are lost. In this case, when the retransmission timer expires, TCP enters the slow start phase and reduces its congestion window to one. Therefore, only a single packet is retransmitted. The second packet will not be retransmitted until the acknowledgment for the first packet comes back. Thus, the total delivery time is \(RTO + RTT_1 + RTT_2\), as shown in Figure 6(c). The slow start mechanism of TCP appears over conservative to our interactive game service, which only sends small
messages intermittently. If both packets were instead retransmitted upon the timeout, the total delivery time would be reduced to $RTO + \max(RTT_1, RTT_2)$. Based on a recent study on the impact of increased TCP initial congestion window [16], such behavior introduces only modest increase in retransmission rate and is therefore quite safe.

4.3.2 Exponential Backoff

The exponential backoff mechanism controls how the retransmissions should be spaced when a packet needs to be retransmitted more than once. The sender will wait until the retransmission timer expires. From our trace-driven TCP replay, we observe that exponential backoff greatly inflates the message delivery time for clients with large RTT. While it is generally believed that this mechanism preserves the stability of the Internet, Mondal et al. [24] have shown that removing the exponential backoff would not induce any stability side-effect. Hence, for the gaming service sending only small messages, it is safe to simply remove the exponential backoff.

4.3.3 RTT variation

TCP constantly updates the retransmission timeout value (RTO). Whenever a new acknowledgment is received, i.e., a new RTT sample is obtained, RTO is updated according to $RTO = SRTT + max(G, K \times RTTVAR)$. The parameter $K$, the multiplier of $RTTVAR$, controls the impact of RTT sample fluctuation on RTO. In standard implementations, $K$ is set to 4. We observe that when the fluctuation is large, the choice $K = 4$ can lead to a large RTO.

For example, one particular client experiences the following RTTs in its trace. For both packets in the $i^{th}$ message, the RTTs are 78ms and 78ms. For both packets in the $(i + 1)^{th}$ message, the RTTs are 360ms and 360ms. For the $(i + 2)^{th}$ message, the first RTT is 391ms, while the second packet is lost. However, the retransmission of the second packet (of the $(i + 2)^{th}$ message) does not occur until nearly 1000ms later, as the fluctuation of the RTTs results in a very large RTO. In the end, the message delivery time exceeds 2200ms.

Intuitively, making RTO calculation more aggressive can help reduce message latency. However, the detailed study from Allman et al. [5] shows that aggressiveness inherently leads to more spurious retransmissions. Therefore, we do not modify the RTO calculation at the moment and leave such optimization for future work.

4.3.4 Minimum RTO

The minimum RTO is an explicit configuration that controls the lower bound of the RTO. The common value used in modern operating systems is 200ms. Vasudevan et al. [30] show that the imbalance between the minimum RTO and low latency can result in poor performance for applications sensitive to millisecond delays; and that removing the minimum RTO improves the performance of such applications and is safe even in wide area networks. From our traces, we observed many clients with RTT values less than 200ms. Due to the small size and infrequency of the game messages, loss can not be recovered via fast-retransmit and has to rely on timeout. Hence, removing the minimum RTO will allow more retransmission opportunities upon loss and thus help reduce message delivery time.

4.3.5 Delayed ACK

The TCP delayed ACK mechanism attempts to reduce the amount of ACK traffic by having a receiver acknowledge only every other packet. If odd number of packets are received, the last packet will not be acknowledged until after a delayed ACK timeout threshold. While this might not be common in a regular TCP transfer with many packets, it could occur frequently in our game scenario. The interactive gaming service ideally calls for a message-oriented protocol, as its messages are small and transmitted intermittently. Hence, a simple modification is to make the delayed ACK aware of the message boundaries. Since all the packets belonging to the same message can be acknowledged with a single ACK, there is little point in delaying the ACK across messages.

4.3.6 Streaming Delivery

TCP transmits all packets in order. When packet loss happens and the retransmission timer expires, TCP reduces the congestion window to one and enters the slow start phase. Newly generated packets will not be injected into the network until earlier packets are acknowledged. This means that when packet loss happens, not only the delivery time of the current message is inflated, but delivery time of the subsequent messages are likely to be affected as well. From our trace-driven TCP replay, we indeed observe such examples. For instance, a message $M_0$ is transmitted at time $t_0$. A single packet loss causes the retransmission timer to expire. The lost packet is retransmitted and TCP also reduces the congestion window. Shortly after, at time $t_1$, a new message $M_1$ is generated. However, at the moment, the congestion window is full, so message $M_1$ cannot be delivered until $M_0$ is acknowledged. As a result, the delivery time of $M_1$ is arbitrarily inflated due to the delay of $M_0$. While in-order
delivery is a useful transport primitive to many applications, gaming services typically build their own message protocol at the application layer. They do not benefit from the in-order delivery and trading the unused feature for performance improvement is desirable.

4.4 Summary

In summary, 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 7, 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. Moreover, as we will later see, such fixes alone are not sufficient to satisfy the game requirement. We are therefore compelled to develop an entirely new transport protocol from scratch. As we will argue in the subsequent section, redundant packet transmission plays a central role in the new protocol. The key challenge, of course, is to design an intelligent scheme that can dynamically adapt to a wide range of varying network conditions.

5 Pangolin Algorithm Design

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. Our primary objective is to ensure that the majority of messages are delivered within the latency tolerance. Specifically, we require the percentage of the messages arriving later than \( T \) be less than a very small target value, denoted as \( \epsilon(T) \).

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.

Given the first order objective, a straightforward solution is to add a fixed number of redundant packets, say \( r \), to each message. If \( r \) is chosen conservatively large, the delay requirement can typically be met. However, such a fixed redundancy scheme has a number of problems: 1) even when there is no loss, it incurs a fixed overhead of \( r \) extra packets per message; 2) 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, a secondary objective is to satisfy the latency requirement with as little overhead as possible.

A dynamic adaptive scheme monitors loss behavior on network links and determines the number of redundant packets to send on-the-fly. If clients are close, it should revert back to pure timeout-based retransmission; if clients are far it should add just enough redundant packets. For a client in the middle range, the latency tolerance period can be divided into stages determined by the round trip time between the client and server. Intuitively, the scheme should favor timeout-based retransmissions in the earlier stages and increase the number of redundant packets in the later ones. To summarize, the dynamic adaptive scheme can be formulated as the following optimization problem – for each individual client we want to minimize the average number of packets sent per message subject to the constraint that the fraction of messages that don’t arrive within \( T \) seconds be less than the target value \( \epsilon(T) \).

5.1 A MDP-Based Solution

In this section, we describe an optimal solution to the above problem based on 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.
5.1.1 An Example

Let’s examine a client with a certain RTT between itself and the server. Suppose that with respect to the RTT, the latency tolerance $T$ can be divided into 3 stages, as shown in Figure 8. When 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 8 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 8 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 8), 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.

5.1.2 Problem Formulation in the 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 8, the action and packet loss determines the evolution of the process. In the terms of Markov Decision Process theory, a mapping from states to actions is called a policy, denoted as $\pi = \{\pi(i, q)\}$. For a given policy $\pi$, the action at each state $(i, q)$ is deterministic (defined by the policy itself). In illustrations similar to Figure 8, 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 endings 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 $\epsilon_T$. Here, the subscript $\pi$ 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 $\rho_{\pi}$.

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

$$\min_{\pi} \rho_{\pi}$$

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

5.1.3 The 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 $\lambda$ and define a combined objective function as a weighted sum of the failure probability and transmission cost, denoted by

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

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 $\lambda$, we solve the modified optimization problem which minimizes $J_\pi(i, k)$ and find the optimal policy $\pi^*$, i.e.,

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

For the optimal policy $\pi^*$ determined by a given $\lambda$, the message failure rate $\epsilon_\pi(i, k)$ 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 $\lambda$ 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 $[8][13]$.

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-trellises rooted at $(i, q)$. Then

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

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

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

Therefore, the problem of finding the optimal policy $\pi^*$ (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.

5.1.4 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 $\lambda$. Then, through bi-section search, the optimal policy can be readily found [8][13].

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 1000 ms. With a RTT quantization step of 10 ms, 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 with less than 4MB in total size. Clearly, all the tables can be pre-loaded by both game consoles and data center servers easily.

5.2 Network Congestion

Any transport protocol that uses FEC to combat packet loss might be problematic in the event of network congestion, where sending extra parity packets can make congestion worse. This is not a major concern for our gaming scenario, in which we transmit small and infrequent messages. But it is desirable to extend Pangolin so that it can serve as a general purpose low latency transport protocol in situations when the emitted traffic is significant. In general, we can run Pangolin on top of a congestion-aware rate control module. Many rate control schemes can serve this purpose. Here, we adopt a
recently developed scheme [23], where the feedback information about both packet delay and loss is used to determine a fair share of the network bandwidth. The scheme strives to maintain low queuing delay even during network congestion.

From Pangolin’s perspective, the information it needs from the rate control module is how many packets can be transmitted at each stage. Let’s call this the transmission budget and denote it as \( B_i \) for stage \( i \). Given the budget, the optimal transmission policy needs to be adjusted to take into account this additional constraint. The problem becomes more prominent when there are multiple messages to be transmitted simultaneously. Messages can no longer be treated independently as they together need to account for the budget constraint. Therefore, the problem of finding optimal transmission policies now becomes (assuming there are \( M \) simultaneous messages):

\[
\min_{\pi} \sum_{m=1}^{M} \rho_{\pi_m}
\]

\[\text{s.t. } \epsilon_{\pi_m} \leq \epsilon(T) \text{ and } \sum_{m=1}^{M} \pi_m(i) \leq B_i,\]

where \( i, B_i \) and \( m \) denote the transmission stage (\( 1 \leq i \leq I \)), the budget constraint at stage \( i \) and the message index (\( 1 \leq m \leq M \)), respectively. \( \pi_m(i) \) is a simplified representation of \( \pi_m(i, q_1, q_2, ..., q_M) \), which denotes the action of message \( m \) at stage \( i \), given the states of all the \( M \) messages.

Unfortunately, solving the above optimization problem is very difficult. The primary reason is that the new trellis representing the state space evolution will now contain an exponential number of states, as all the combinations of the states from individual messages have to be considered. To simplify the optimization and obtain a low complexity solution, we make the following two approximations. First, we assume the Lagrangian multiplier \( \lambda^* \) discovered without the bandwidth constraint represents a satisfactory trade-off between the message delivery latency and the transmission cost. Hence, we modify and incorporate the latency constraint into the objective function, which in turn becomes a combination of the latency and the cost, weighted by \( \lambda^* \). Second, instead of trying to solve the optimization problem across all the stages, we focus only on the current stage and drop the bandwidth constraint on future stages. We rely on the continued execution of the optimization to rein in future actions. That is, when a future stage become current, the optimization at that stage will ensure all the actions then satisfy the bandwidth constraint. Therefore, the modified optimization problem becomes:

\[
\min_{\pi} \sum_{m=1}^{M} (\epsilon_{\pi_m} + \lambda_m^* \rho_{\pi_m})
\]

\[\text{s.t. } \sum_{m=1}^{M} \pi_m(1) \leq B_1,\]

where \( \lambda_m^* \) corresponds to the optimal policy of message \( m \) without the bandwidth constrain\(^1\).

This turns out to be a classic knapsack problem – how to allocate the transmission budget \( B_1 \) among the \( M \) messages so that a total cost can be minimized. Hence, an optimal solution can be readily derived using dynamic programming, as long as the total cost is well-defined given a specific allocation. To this end, let’s denote \( \{ b_m \} \) as an allocation, where \( b_m \) packets are transmitted by message \( m \). Then, the total cost can be represented as:

\[
C = \sum_{m=1}^{M} (\lambda_m b_m + \sum_{q'=0}^{q} p(q'|q, b_m) J_m^*(2, q'))
\]

where \( J_m^*(2, q') \) is the optimal cost of message \( m \) at stage 2 given state \( q' \). Since the bandwidth constraint is dropped for all the future stages (include stage 2), \( J_m^*(2, q') \) can be calculated independently for each message without the bandwidth constraint and thus the same as in Equation\(^5\). Therefore, to facilitate the computation, similar to before, the optimal costs \( J^*(i, q') \)’s can be pre-computed and stored in look-up tables. Since there is a one-to-one mapping between a cost and an action, storing the cost tables doubles the memory consumption\(^2\).

6 Protocol Design and Implementation

Based on the above adaptive FEC scheme, we have implemented the Pangolin transport protocol, to ensure that the 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 design and implementation.

\(^1\)\( \lambda_m^* \)’s can be different when messages of different sizes are transmitted together.

\(^2\)Since messages can arrive asynchronously, stage 2 of an early message can occur at the same time as stage 1 of a new message. The look-up tables have to include all the future stages, not merely stage 2.
6.1 Protocol Design

There are two types of packets in Pangolin: data packet and ACK packet, as shown in Figure 9. Both types packet share the same packet header. In the packet header, the first 16 bits indicate the protocol version. The next field is packet length, which records the length of the entire packet, including that of the packet header, a data header or an ACK header, and a payload length. It has 16 bits, so the maximum length of a packet can reach 64KB. However, considering the size of maximum transmission unit (MTU), it is typically no more than 1500B. The next 16-bit field is packet type, which distinguishes data packet from ACK packet. The next 24-bit field is reserved. During experiments, this field is used to collect packet level statistic information.

A message is broken into a sequence of data packets. In each data packet, the packet header is followed by a data header, which records information related to the original message. The 32-bit message sequence number can identify all the packets belonging to the same message. Packets are discriminated by packet ID. Field N and K contain FEC information, where N indicates the total number of packets in the message and K indicates the number of packets needed to recover the message.

Besides the packet header, an ACK packet has an ACK header, which acknowledges a specific data packet identified by the message sequence number and the packet ID. The 56-bit acknowledgment number field is a bit vector representing packets received in the past. It is used to support the implementation of selective acknowledgment.

6.2 Implementation

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 architecture of the Pangolin stack is illustrated in Figure 10.

The interface module is in charge of interacting with upper layer applications. Outbound messages are entered into the outgoing message queue and wait to be processed by the scheduler. Inbound messages are assembled and put into the incoming message queue for the applications. The interface can be invoked via both synchronous and asynchronous APIs.

The flow manager module maintains per flow status. For each flow, identified by (IP, port), it creates a transmission control block (TCB) and keeps all the important information, such as the number of transmissions and losses, the message deadline, a sliding window, and other statistics. The sliding window controls the number of messages on the fly, i.e. those still in transmission and not completely acknowledged. The number of transmissions and losses, together with the message deadline, are used as inputs to the adaptive algorithm, which is described in the previous section and implemented in the FEC module. The output of the FEC module is the optimal action for each transmission stage.

The scheduler is the core engine of Pangolin. It processes the messages from the outgoing message queue, encodes them using the FEC module, and enters prepared
packets into the outgoing packet buffer. Also, it decodes incoming packets from the incoming packet buffer, and enters assembled messages into the incoming message queue. Moreover, it periodically triggers the flow manager to execute background tasks, such as retransmitting timeout packets, cleaning up obsolete TCBs, and so on.

The IO engine module is in charge of sending and receiving individual packets. For high concurrency game servers in the service data center, I/O can easily become a performance bottleneck. In Pangolin, the IO engine is implemented using I/O completion port (IOCP) [19], the most efficient I/O mechanism on Windows supporting the highest throughput and scalability.

The core of the Pangolin stack contains about 7000 lines native C++ and about 500 managed C++ code to support C# applications.

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 \( \pi^*(q_2 = k_2) = k_3 + 1 \), 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 calculated as \( (\rho^* - k)/k \), which is \( (4.16 - 4)/4 = 4\% \) and is very small.

Figure 11 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. 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 12 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

---

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

7.3 Latency Under Bandwidth Constraint

In this section, we evaluate the performance of Pangolin with the bandwidth constraint. We compare the Pangolin policy, as illustrated in Section 5.2, to a naïve policy, which transmits messages on a first-come-first-serve (FCFS) basis. For each message, FCFS transmits as many packets as dictated by the optimal policy computed without the bandwidth constraint, up to the actual bandwidth limitation.

We design the following experiment to compare the two policies. The round trip time is again 250ms and the target latency is twice the RTT at 500ms. Messages are created at a constant rate, about one message per round trip time. The message size varies with a uniform distribution between 2KB to 12KB. The packet loss rate is fixed for each experiment, but varies across experiments from 2% to 20%. For each experiment, we artificially set the bandwidth constraint to a constant value. Since we are sending one message per RTT, it makes sense to calculate the bandwidth requirement to achieve the 99.9% success rate for each message (following the optimal policies calculated in Section 5), and set the bandwidth constraint as the average of the required bandwidth across all the messages.

With such a setup, during each experiment, we transmit 36,000 messages and examine the percentage of messages that are successfully delivered within the 500ms target latency. We repeat each experiment 10 times and report the average across the 10 experiments. Figure 13 compares the performance between FCFS and Pangolin across a wide range of packet loss rates. It is clear that Pangolin works much better than the naïve FCFS policy. Pangolin achieves more than 99% success rate almost in all the cases (except for loss rate of 12%, the success rate is 98.9%), while the success rate of FCFS vary widely from 70% to 90% and is never as high as Pangolin. From another perspective, the comparison also highlights the effectiveness and importance of a more elaborated adaptive scheme.

We remark that Pangolin is unable to achieve the ideal success rate of 99.9% during all the experiments. This is understandable as (i) the constant bandwidth constraint takes an average value of the bandwidth requirements across all the messages, which can affect messages of bigger size; and (ii) high-delay early messages can affect later messages given the constant bandwidth constraint.

Note that the bandwidth constraint choice is rather arbitrary here. We believe it is reasonable as the main evaluation metric is not how well a particular policy performs, but rather the relative comparison between the two policies.
7.4 Trace-Driven Emulation

7.4.1 Latency

<table>
<thead>
<tr>
<th>Latency (ms)</th>
<th>TCP</th>
<th>Retrans.</th>
<th>Fixed FEC</th>
<th>Pangolin</th>
</tr>
</thead>
<tbody>
<tr>
<td>95%</td>
<td>219</td>
<td>219</td>
<td>204</td>
<td>219</td>
</tr>
<tr>
<td>99%</td>
<td>566</td>
<td>469</td>
<td>328</td>
<td>412</td>
</tr>
<tr>
<td>99.5%</td>
<td>828</td>
<td>703</td>
<td>469</td>
<td>581</td>
</tr>
<tr>
<td>99.9%</td>
<td>2352</td>
<td>1501</td>
<td>984</td>
<td>938</td>
</tr>
<tr>
<td>Overhead</td>
<td>-</td>
<td>3.9%</td>
<td>150%</td>
<td>6.1%</td>
</tr>
</tbody>
</table>

Table 3: Latency Performance Comparison in the Wild.

![Figure 14: Latency Performance Comparison in the Wild.](image)

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\(^5\). 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 packets are added, i.e., 5 packets are sent in total\(^6\).

Table 4 shows the comparison results. The most important performance metric is the latency at the 99.9-per 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 14 shows the CDF of the four schemes, where the right plot is a zoomed view of the left one. 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.

7.4.2 Cost

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

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.

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...
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 [27]. 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 [29]. 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. [26] 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 [25, 28] or retrieving from multiple sources [12]. Efficient codes are also developed to reduce computation complexity, such as digital fountain and online codes [11, 22]. 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 the game, 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 theoretical analysis and trace-driven emulation demonstrate that Pangolin minimizes the maximum latency (for nearly all users) while keeping the overhead negligible. Pangolin has already been incorporated into the latest Xbox SDK, which was released in November, 2010 and is now powering concurrent messaging for hundreds of thousands of Xbox clients.

References

[1] 1 vs. 100 (Xbox 360). In Wikipedia.
[21] Liew, J. Why the economics of social gaming are so attractive to investors, December 2009.