PSP : A Persistent Streaming Protocol for Transactional Communications[*]

Kien A. Hua, Ning Jiang, Rui Peng, Mounir A. Tantaoui
School of Computer Science
University of Central Florida
Orlando, Florida 32816-2362
{kienhua, njiang, rpeng, tantaoui}@cs.ucf.edu

Abstract In a distributed environement, many applications require reliable and fault-tolerant communications. While most transactional techniques rely on either a 2-phase commit protocol or checkpointing, neither of these solutions is suitable for large batch processing applications. In this paper, we describe the Persistent Streaming Protocol (PSP) that guarantees fault-tolerant communications between a sender and one or multiple receivers. An application that uses the PSP protocol sends data only once, and they will eventually reach the destinations regardless of system or network failures. We first describe the PSP protocol and then the prototype implementation.

Keywords: Fault-tolerant, communication protocol

I. INTRODUCTION

Centralized systems are becoming an "antique curiosity." Most of today's applications are running on a distributed system. Many of these applications have a need to send a large amount of data from a source to multiple destinations in a transactional way. That is, all receivers must eventually receive all of their data in order to commit the transactional transmission. Such applications can be grouped into two major categories: the *distributed notification style* and the *asynchronous notification style* [3]. In the former style data sources are synchronized, whereas they are not in the latter. In both cases, data are distributed and strong recovery is needed. Many such applications would involve an initial gathering of the data; and then a batch delivery at some precise time. One example is the distributed payment atomicity in e-commerce discussed in [5]. It is highly desirable that the orders are guaranteed to be successfully delivered to the target e-commerce system even in the case of a system or network failure. Another example is the banking industry where many banks scan large number of financial documents (e.g. cheques and deposit slips) and store the images of those documents to a distributed archive consisting of multiple branches/data centers in a transactional way. We note that in the above examples the input of the system is large number of streamed data items that are generated on-the-fly. In many other applications, the purpose can be to transmit files that already exist in the persistent storage to multiple destinations in a fail-safe way. Currently, no communication protocol has been developed to support all these critical data processing environments. We cannot simply rely on TCP since TCP cannot handle situations where system crashes or problems arise in the network.

The traditional solution to the above problem is to implement the 2-phase commit protocol. That is, the sender streams all the data items to their appropriate receivers and waits for acknowledgement from each of the receivers in Phase 1. In Phase 2, if all the acknowledgements are positive, the sender informs all the receivers to write the data to their permanent records, and reply to the sender with a commit message in Phase 2. The sender can now commit the transactional transmission. This approach, designed for short-live transactions, is not suitable for many batch processing applications that involves transmitting a large amount of data to multiple destinations. If any one of the machines crashes or problems occur in the network, the transaction would have to be aborted and redone. This recovery strategy is unacceptably expensive. One way to reduce this cost is to apply check pointing. In this case, we can rollback to the previous check point if problems arise instead of having to abort the entire transaction. Check pointing, however, incurs overhead. In any case, all of these solutions have one undesirable effect - system instability at one location would affect the normal operation of all other destinations.

In this paper, we consider a *Persistent Streaming Protocol* (PSP) on top of TCP, at the application level, to address the aforementioned issues. With PSP, data that are successfully submitted to the system are guaranteed to eventually reach their destinations regardless of the system condition. If a receiver crashes at the middle of a transmission session, it does not affect the normal operations at other locations. When the failed location is up again, it will continue to receive its data items from where it left off. Thus, rollback and redo are no longer needed in the PSP environment. To the best of our knowledge, this is the first communication protocol designed to address the transactional issues.

PSP differs from the *Transaction Transmission Control Protocol* or T/TCP [4] in the sense that the latter is a transaction-oriented protocol based on a minimum transfer of segments. Another similar work proposed in [2] modifies HTTP to transport multiple requests over each TCP connection. The persistent-connection HTTP or (P-HTTP) [2] avoids the excessive round trips in the current HTTP

[*] This research is partially funded by ImageSoft Technologies, a subsidiary of Fiserv Inc.

protocol. However, none of these works address the problem of fault-tolerance.

PSP is also different from the File Transfer Protocol (FTP) in several aspects, firstly FTP is not a streaming protocol and data using FTP should have already been stored. Secondly, and as we shall discuss in section III, FTP do not provide fault-tolerant issues as PSP. PSP can therefore be used as a middleware layer for integrating software subsystems to build transactional systems.

An application using PSP can send data to one or many destinations. When sending to many destinations, PSP follows the multi-unicast model as opposed to multicast. The reason for such choice is primarily due to limited deployment of multicast hardware on the Internet due to several fundamental concerns [1] [6].

The remainder of this paper is organized as follows. We introduce the PSP in Section 2. In Section 3, we present a prototype and our experimental study. We give our concluding remarks in Section 4.

II. PSP DESCRIPTION

PSP is responsible for application registration and session management. It also provides the *application programming interfaces* (*APIs)* for applications to submit and receive data as well as retrieving session status. Finally, PSP guarantees the reliable delivery of data under the presence of failure at multiple points of transmission. We describe these operations in this section. In the description, we refer to the *sender-side* PSP layer as PSP_S, and the *receiver-side PSP* as PSP_R.

1. *Application Management*

A PSP_R allows more than one receiver application to receive packets simultaneously while each receiver application may receive data from multiple sources concurrently. This calls for an application management scheme in each PSP_R.

A receiver application should register with a local PSP_R before it can use PSP protocol to receive data. A PSP_R keeps the information of all the registered applications in a volatile table *AppInfo* which contains the following three columns: **receiver ID**, **receiver Port**, and **status**. The last parameter has either a *Registered or a Listening,* value. When an application calls the **Register** *API*, corresponding PSP_R assigns a local *ID* for the registering application and inserts the application information into the table.

Table *AppInfo* does not need to be persistent. In fact, if a PSP_R fails, all the attached applications will get disconnected and wait for the PSP_R to be up to register again. A registered application can start listening for incoming connections on a specific port by calling the **Listen** *API*. Upon receiving the listening request from the receiver application, PSP_R updates the *AppInfo* table by recording the listening port and setting the *Status* to *Listening*. The application will block until a new sender connection comes in. The application should spawn a thread for each connection before it goes back to listening state. In this manner, one receiver application can receive data from multiple senders concurrently.

2. *Session Establishment*

Before a sender application can start transmitting data to multiple destinations, it should first establish a connection with each of the destination receivers. Hereafter, we refer to the whole data transmission process as a *data transmission session* (or *session* in short) and each individual connection between the sender and a destination a *sub-session*. The **Connect** API is designed to perform session establishment operations.

The sender application passes information of each destination receiver to the local PSP_S through the *API*. PSP_S utilizes this information to contact each destination PSP_R. Connection with each receiver is established through a two-way handshake process. Consider a particular pair of PSP_S and PSP_R. First, PSP_S transmits information of both the sender and the intended receiver application to PSP_R, which verifies whether the intended receiver application has already registered or not. If the receiver application has registered and is in Listening state, PSP_R will reply a connection success packet to PSP_S and the handshake ends successfully. Otherwise, a connection failure packet is sent to PSP_S. If all the PSP_R's reply with connection success packets, PSP_S allocates resources for the session and returns the *session ID* to the sender. The session *ID* will be used by the sender application during data submission and is unique within each location. PSP_S can start accepting data from the sender application only if connections are successfully established with *all* the destination receivers. If, however, any of the connection attempts fails, the *API* notifies the sender application, which has to wait until all the receivers are ready to accept data.

On the receiving end, PSP_R does not allocate resources for a session until it receives the first data packet of a session. We note that the fact that a data packet is received by a PSP_R indicates that the sender has successfully established connections with *all* the receivers. Only at this time is it safe for the PSP_R to allocate resources for the session locally. Upon receiving the first data packet from *PSP_S*, *PSP_R* invokes a session setup procedure if the status of the requested receiver is *Listening* .

- PSP_R stores the information of this session in a persistent table *ConnInfo*, which contains a *session ID*, PSP_S *IP* address and receiver *ID*. The [session ID, PSP_S IP] pair uniquely identifies a sender application. *Session ID* or *PSP_S IP* alone may be insufficient because one PSP_R may have sessions from different PSP_S with the same *session ID* and multiple sender applications running on the same PSP_S have the same *IP*. If we suppose that PSP_R fails before sending an acknowledgement (which is stored persistently) to some PSP_S, PSP_R needs to know where to send the acknowledgement after it restarts, therefore *ConnInfo* table has to be persistent.
- PSP_R creates a memory buffer for this session to hold the incoming data.
- PSP_R allocates a persistent block for the receiver application to store the latest acknowledgements and return the address to the receiver application along with

the session information (*session ID* and *PSP_S IP address*).

If any of the above steps fails, PSP_R will send a session error packet to PSP_S, which will refrain from sending data to the corresponding receiver application and attempt to recover the sub-session later on. *Figure 1* gives an example of a typical connection using PSP.

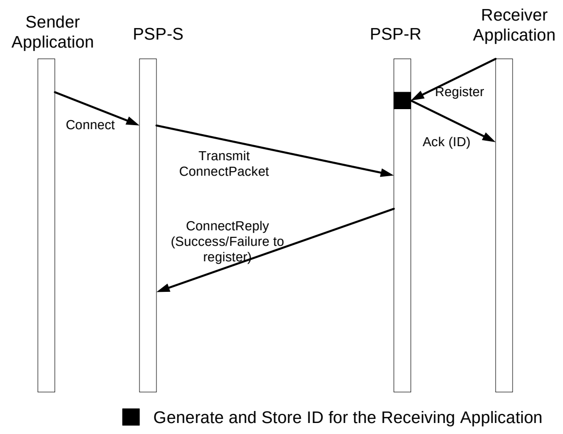
Sender Application
PSP-S
PSP-R
Receiver Application
Register
Connect
Ack (ID)
Transmit ConnectPacket
ConnectReply (Success/Failure to register)
■ Generate and Store ID for the Receiving Application

Figure 1. Typical connection with PSP

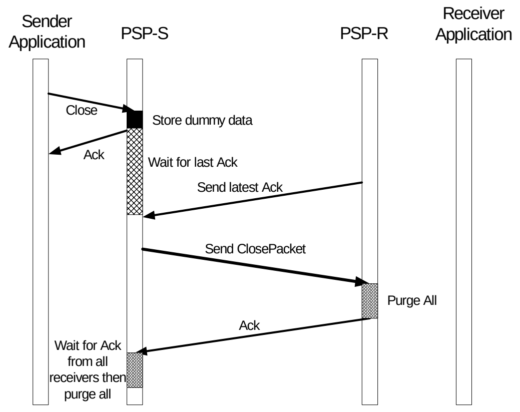

Figure 2. Closing a session in PSP

3. *Session Close*

When a sender successfully submits all its data to PSP_S, it invokes the **Close** API to inform PSP_S that there is no more data to be transmitted. PSP_S will then mark the session as closed internally. However, PSP_S and various PSP_R corresponding to the receivers cannot purge the resources of the particular session until all the data are successfully received and processed by all the receiver applications. This is achieved by retrieving the acknowledgement information received from each individual PSP_R. If PSP_S finds that all the data have been successfully acknowledged, it sends a session close packet to each individual PSP_R, which clears the resources corresponding to the session and acknowledges PSP_S. PSP_S cannot purge resources for a particular session unless it has received confirmation packets from all the destinations of that session. In case some destination fails to reply, PSP_S will periodically retransmits session close packets to all the un-acknowledged PSP_R's until it receives all confirmations. *Figure 2* gives an example of closing a session in PSP.

4. *Data Submission and Delivery*

The **SendData** API is designed for the sender application to submit data to its corresponding PSP_S. In all cases, the size of a submitted data item must be less than a system parameter *F*. Otherwise PSP_S will report an error and quit. Information of the submitted data is maintained in a persistent data table (PDT). The format of a PDT entry is <*DataID*, *data*, *pFlag*, *fInfo*, *offset*>. There are two cases we need to consider. First, the input data items are from a data stream generated on-the-fly. In this case, data items must be made persistent to facilitate the recovery process. In the SendData API, user explicitly specifies that the particular data item should be pooled. When PSP_S receives such a data item for a particular session, it creates an entry in PDT, fills in the *DataID* field with a newly generated data ID and stores the data item in the *data* field. Moreover, it sets the *pFlag* field to TRUE and leaves the *fInfo* and *offset* field to 0. In most cases, PSP_S is able to allocate enough resources to store the submitted data. Otherwise, *SendData* fails and the sender application has to call the Reconnect API to recover the session after problems are resolved. Details of such recovery are further discussed in Section 6. In the second case, the input are existing files. The application is in charge of splitting files into pieces less than or equal to *F* and submit them to PSP_S. PSP_S creates an entry for each item in PDT and generates a dataID for each piece. The *pFlag* field is set to FALSE. The application must also provide the path information and the offset of the item in the original file for each item. These information is stored in the *fInfo* field and the *offset* field respectively. In both cases, the data item is also pushed into a memory buffer allocated by PSP_S after session establishment. A data transmission thread is created for the session and it constantly checks the memory buffer and transmits the data items and their IDs to the destinations. As a result, the overall performance is greatly improved since disk I/O is saved for each data item. By keeping data on persistent storage, even if failure occurs during data delivery, we are able to retransmit the data later on when the system recovers. Once PSP_R receives a data packet from PSP_S, it extracts the data and puts the tuple <*data ID, data*> into the memory buffer allocated to the destined receiver application.

5. *Data Reception and Acknowledgment*

Once a connection is established in PSP_R, the receiver application receives the acknowledgment address along with the *session ID* and *PSP_S IP* address. The receiver can then call the **Receive** *API* to retrieve data from the corresponding data buffer maintained by PSP_R. The *Receive* API retrieves data block (<*data ID, data*>) one at a time and will block the application if the buffer is empty.

Receiver application acknowledges the sender by calling the **Acknowledge** *API*. Usually a receiver application will acknowledge the receipt of data only after fully processing the data. In our sample application, it acknowledges only after it successfully stores the data into local disk.

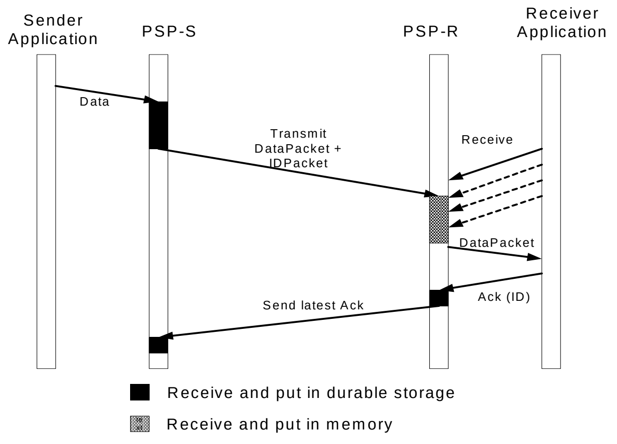

Figure 3. Transmission and reception

Upon receiving the acknowledgement from an application, PSP_R saves it in the pre-allocated persistent storage. Acknowledgements are periodically transferred back to the appropriate PSP_S, which utilizes the information to perform disk space reclaim and session recovery. Data items that have been acknowledged by *all* the receivers are immediately purged from the PDT to allow space for more data. We discuss session recovery in the following subsection. *Figure 3* illustrates the transmission and reception in PSP.

6. *Automatic Recovery*

During data transmission, failure can occur at the sender, network and/or the receiver side. PSP incooperates an automatic recovery manager to deal with situations when failure occurs during data transmission. We consider two scenarios where recovery is performed. The first scenario copes with situations where failure occurs at the network and/or receiver side. The latter one happens when sender side failure occurs. In both scenarios, recovery is performed on sub-session basis. In the current implementation, recovery is always started by PSP_S.

First, it is possible that some receivers (PSP_R and/or receiver application) or the network fails during data transmission. The recovery manager periodically verifies whether a sub-session needs to be recovered. We refer to the time interval that the recovery thread is invoked as the *recovery interval*. In both cases, the recovery manager checks the session specific information maintained in the *SessionInfo* table on persistent storage. The table consists of two parameters: (1) **SentID** that describes the data to be sent and is updated whenever a data item is sent to the destination. Similarly, (2) an **AckID** that describes the acknowledgement received for a specific data and is updated whenever an acknowledgement is received. Moreover, for each parameter, there exists a parameter that describes a previous state of sending or receiving data and acknowledgments respectively, namely **previousSentID** for *SentID* and **previousAckID** for *AckID*. When a session is initiated, *prevSentID*, *SentID*, *prevAckID*, *AckID* are all set to –1. Each time the background recovery thread starts, it checks the aforementioned fields of each session. If any of them is –1, it simply copies the value of *SentID* (*AckID*) to *prevSentID* (*prevAckID*), respectively, and goes back to sleep. Otherwise, it verifies the following condition :

(prevSentID==SentID) AND (prevAckID==AckID)

If the condition is satisfied, it means that no data is transmitted and no acknowledgement is received during the recovery interval. Thus, the sub-session is "silent". A sub-session becomes silent due to the following two possiblities. First, PSP_S has finished transmitting all the submitted data items for that sub-session and is waiting for more data. Recovery is not necessary in this case. Second, the particular sub-sessions is encountering failure in any of the sender, network and receiver sides. The recovery manager first tries to establish a connection with the corresponding PSP_R. If it succeeds the PSP_R will reply with the failure point information (i.e. latest acknowledged data item) for that sub-session and the recovery manager will transmit the pending data to the receiver.

In the second scenario, a sender application invokes the **Reconnect** API after a failure occurred at the sender side. The sender application does not need to maintain any information of the failure point. PSP will automatically figure out the necessary recovery information for the particular session based on the *SessionInfo* table. At this point, the application should submit data starting from the latest allocated data ID to PSP_S. However, it is possible that data submitted before the last system failure are still pending to be transmitted to various PSP_R's. The recovery manager is thus invoked to recover those data. If it successfully transmits all the remaining data for at least one of these sub-sessions, it informs the sender application and enables the memory buffer. In this case, higher performance can be achieved since PSP_S can transmit the newly submitted data directly from memory. Sub-sessions that cannot be recovered have to wait for the next background recovery session and extra disk reads will be performed for the submitted data items.

To further improve recovery performance, a cache is introduced to store data items that have been transmitted by the recovery manager. This can significantly improve performance in the situation where a sub-session is experiencing continuous failure and needs to be recovered for multiple times or for sessions that have multiple sub-sessions to be recovered.

III. EXPERIMENTAL STUDY

Our prototype is being tested for a distributed archive product at ImageSoft Technologies. The distributed archive is commonly used in the banking industry to archive large number of images of finance documents (e.g. cheques, deposit slips) in multiple branches/data centers. In the test environment, a stream of approximately 2000 finance document images are captured on-the-fly and transmitted from a Sun Enterprise 250 server with two 400-MHz UltraSPARC-II CPUs running Solaris 8.0 to one or more receivers running on PCs of Intel Pentium II 400MHz CPUs with Red Hat Linux 8.0 installed. The size of images range from several Kilobytes to several Megabytes. At each receiving end, a receiving application stores the received images into a specified directory.

We tested PSP with the following scenarios. The first test case confirmed that a sender could establish multiple communication sessions simultaneously. Each could involve more than one receiver. The second test demonstrated that a receiver could be in session with more than one sender. These two cases were designed to test the different multi-unicast scenarios under PSP. We then tested the fault tolerance capability of PSP. In all the experiments, the recovery interval was set to two minutes. We physically removed the line from a computer (i.e., communication failure) or from a power source (i.e.., power failure) and made the following observations in this study: (1) A disconnection of the communication link to *Receiver 1* did not affect the data delivery to *Receiver 2*. Furthermore, Receiver 1 could resume the transmission when the broken link was physically reconnected. (2) The same behavior was observed when we unplugged the power line of *Receiver 2*. (3) When we disconnected the sender from the power source, the whole transmission paused. However, the session recovered and resumed the data transmission after we replugged the sender to the power source. In all cases, every image arrived orderlly and correctly at the intended destinations.

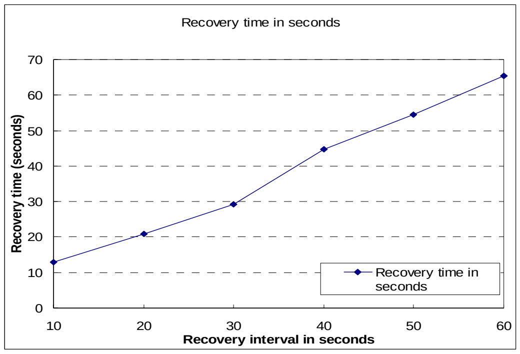

Figure 4. System recovery time

We depict the relationship between the time that takes the system to recover from a failure and the recovery interval in Figure 4. In the experiment, we had one sender application sending images to a remote receiver application. We crashed the PSP_R in the middle of transmission and restarted it sometime later. We recorded the time when the receiver application was restarted and the time when its data reception was resumed, and the recovery time was calculated as the difference between the two. We varied the recovery interval from 10 seconds to 60 seconds and recorded the results in Figure 4. In the figure, the X-axis represents the recovery interval and the Y-axis represents the recovery time. We observe that the recovery time is approximately proportional to the recovery interval. In the current distributed archive environment, we set the recovery interval to be 2 minutes.

To test the time it takes to recover a session, we performed another test. In the test, the sender transmitted 200 images to two receiver applications. We unplugged the network link after 50 images were successfully transmitted to receiver applications. As a result, the data were continually pooled in the PDT whereas the receiver applications were unable to receive any data. After all the images were stored in the PDT, we plugged the network link back and measured the time it took to recover both sub-sessions. The recovery time for a particular sub-session was measured as the difference between the time when the recovery thread attempted to reconnect to the receiver application and the time the last data item was successfully acknowledged by the receiver application. Table 1 demostrates the recovery time of both sub-sessions. From Table 1 we observe that the recovery time for the second sub-session was much less than the first one. This is due to the employment of the recovery cache. During the recovery of the first sub-session, data items were stored in the cache. Consequently, the recovery of the second sub-session did not involve any disk operation and achieved a 69% improvement in terms of recovery time.

Table 1. Recovery time of sub-sessions

Sub-session ID	Recovery time (in seconds)
1	143
2	45

IV. Conclusion

A transactional communication session is an atomic communication operation that must deliver the dataset to its destination in its entirety, otherwise any partial data transmission must be undone as if the communication session has never occurred. Transmission protocols in the transactional framework can result in a session abort, and therefore require retransmission of the data.

In this paper, we proposed a new protocol that deals with fault-tolerance issues. The Persistent Streaming Protocol (PSP) session never aborts; automatically recovers failed sessions efficiently and retransmissions are never needed. It is therefore suitable for a wide range of applications including long-live communication sessions (e.g., ftp, batch processing, etc.)

References

[1] Yang-Hua Chu, Sanjay G. Rao, and Hui Zhang, “A case for end System Multicast,” in *ACM SIGMETRICS*, 2000, pp. 1-12.

[2] Jeffrey C. Mogul. The Case for Persistent-Connection HTTP. SIGCOMM ’95 Cambridge, MA USA pp. 299- 313.

[3] Yoshitomi Morisawa, Koji Torii. An Architectural Style of Product Lines for Distributed Systems, and Practical Selection Method. 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering, Sep 2001. Vienna, Austria.

[4] http://rfc-1644.rfclist.org

[5] Heiko Schuldt, Andrei Popovici, Hans-Jorg Schek. Automatic Generation of Reliable E-Commerce Payment Processes. Proceedings of the 1st International Conference on Web Information Systems Engineering (WISE’2000) pages: 434-441, Hong Kong, China, June 2000.

[6] Duc Tran, Kien A. Hua, and Tai Do, “ZIGZAG: An Efficient P2P Scheme for Media Streaming,” IEEE Infocom 2003.