Multimedia Broadcasting System: An Implementation of GRUM With Recommended Technical Improvements for GRUM Communication Models

Lim Tong Ming, Universiti Tunku Abdul Rahman, 9 Jln Bersatu 13/4, PJ, Selangor, Malaysia.

tongminglim@gmail.com

Chin Tek Min, haleychin@gmail.com

Teh Chia Ching, sakuragi_003@yahoo.com

Chong Khong Mun, ckm3812@yahoo.com

Abstract

Multimedia Broadcasting System (MBS) is a distributed media software system that provides reliable broadcasting services using GRUM over the Internet. The proposed Multimedia Broadcasting System ensures that all media files broadcasted by the sender will be received by the receivers through copper-based public switching network which provides unreliable low bandwidth communication services. This paper introduces GRUM as the reliable unicast/multicast API for the communication layer between the sender and receiver to achieve the reliable broadcasting services. The MBS also proposes a ***Split-and-Join Later*** *algorithm to improve on the GRUM's communication models for the broadcasting of large media files over the unreliable Internet connection in the Malaysian public switching network environment.*

1. Background, Motivations and Goals

Over the past twenty years, there are lots of new applications and technologies emerged due to the advancement of network infrastructure and technology. With the latest development in the networking technology and higher demand for multimedia streaming services, higher quality of service and more reliable streaming are critical ingredients in the broadcasting industry. Multimedia streaming is activities which allow video and sound to be sent continuously and concurrently to one or many users by service providers. For countries that have broadband connection, multimedia streaming services can be delivered with high degree of quality of service. However, for countries that are lack of broadband connectivity, multimedia streaming is a great challenge due to network traffic congestion and unstable connection. These issues have resulted in poor streaming performance. Although existing streaming multimedia technologies such as peer-to-peer streaming and stream-on-demand have been serving the users for years, but these technologies do not guarantee ordered and reliable delivery services in a copper-based public networking infrastructure like Malaysia. As a result, this will lead to the media files corruption and low level of quality-of-service.

The project examined in this paper has set few objectives to be achieved. They are as follow:

1) To design, implement and prototype a **Multimedia Broadcasting System** (MBS) using the GRUM API [1].
2) To test and evaluate the reliability and performance of the multimedia streaming services.
3) To carry out detail examination and discussion on the GRUM's communication model in order to study potential enhancements to the current GRUM's communication models so that the performance of the MBS can be improved.
4) To propose enhancements to GRUM to improve on its capability to handle very large media files without relying entirely on the heap space of the Java Virtual Memory.

2. Review of Research Works

SRM (Scalable Reliable Multicast) and RMA (Reliable Multicast) consists of various kinds of recovery mechanism. In general, the recovery strategies consist of a prioritized list of recovery servers/receivers (clients) and/or randomly organized. Recovery requests are sent to the recovery clients on the list one-by-one until the recovery effort is successful. Danyang Zhang, Sibabrata Ray, Rajgopal Kanna, S.Sitharana Iyengar, however, had proposed a polynomial time algorithm [2] for choosing the recovery strategy with low recovery latency without spending much bandwidth and performs 20% greater than the other two algorithms.

Reliable probabilistic multicast (rpbcast) [3] proposed by Qixiang Sun, Daniel C.Sturman, and Tatsushiro Tsuchiya and Tohru Kikino is a hybrid of centralized and gossip-based approaches as the recovery mechanism in a large systems. Qixiang Sun and Daniel C. Sturman [3] used Log-based Receiver-reliable Multicast (LBRM) and Bimodal Multicast (pbcast) as the basis and propose a more reliable version of reliable multicasting by mixing LBRM and pbcast. However, the traditional approaches do not scale well due to centralized recovery mechanisms and excessive message overhead. Rbpcast, uses gossip as the primary retransmission mechanism and only contacts loggers if gossips fail. Large groups of active senders are supported using negative gossip that specifies those messages a receiver is mussing instead of those messages it received.

Expanding ring search (ERS), is a mechanism for building a scalable and reliable multicast tree based system whereby the receivers are organized in an ACK tree. However, ERS has a poor scalability, strong dependency on the multicast routing protocol and the need for bidirectional multicast capable networks which cause ERS become the Internet standard mechanism. According to Christian Maihofer, author for Scalable and Reliable Multicast ACK Tree construction with the Token Repository Service [4, 5], shortcomings raised by the ERS in the ACK tree for reliable multicast can be resolved by Token Repository Service (TRS) which is based on a token repository and a modification of ERS. The TRS store tokens, which represents the right for a joining node to connect to a certain parent node in the ACK tree.

In short, the approaches studied in [2,3,4,5] can be critically commented and compared as in **Table 1.0**.

Criteria studied	**LBRM**	**Pbcast**	**Rpbcast**
Protocol	Log-based	gossip-based	hybrid protocol
Link utilization	No	Better link utilization	Better link utilization
Distribution load	Plenty of retransmission among nodes	Balance	Balance
Local buffer	Not require	Buffer require	Buffer require
Reliable log server	Need a reliable log server	Not require	Not require
Latency (from sender to all receivers)	High when high send rate	Middle latency	Random re transmission selection, sometimes may not succeeded – but low latency
Recovery mechanism	Push-base recovery	Push-base recovery	Pull-base recovery
Message overhead	constant difference in overhead between rpbcast and LBRM is due to additional information in rpbcast's gossip messages so LBRM will increase due to an additional membership protocol	approximately linear with the number of senders	-
Message overhead types	consist of acknowledgments, periodic heartbeats, and retransmission requests	gossip messages and retransmission requests	consists of gossips, garbage collected notification and acknowledgements

Table 1.0: A detailed comparison of the three (3) approaches studied in [2,3,4,5]

3. Guaranteed Reliable Asynchronous Unicast and Multicast

Guaranteed Reliable Asynchronous Unicast and Mulicast (GRUM) architecture applied the Layered Architecture for the entire component. GRUM engine consists of few modules; there are GRUM.util, GRUM.API, GRUM.Client, GRUM.Node, GRUM.MulticastTransport and GRUM.UnicastTransport. The GRUM.Util module provides a common utility framework for the entire GRUM architecture. GRUM.API module publishes all the necessary functionalities which can consume by others. GRUM MulticastTransport and UnicastTransport are the modules that provide the communication services either Multicast or Unicast among all the computers. GRUM.Node module is designed for handling the membership for every computer that joins the group. Node membership will be monitor by the coordinator for maintain the heartbeat and session for each node. GRUM.Client module provides basic commands such as send, sendto, createChannel, setMemberListener, setMulticastListener, setUnicastListener and so on.

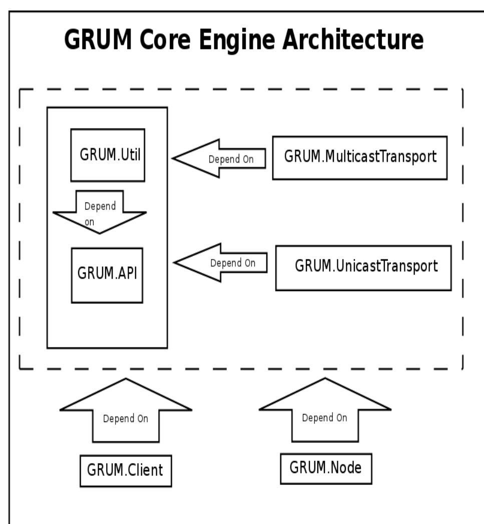

Figure 1: GRUM Architecture.

3.1 Detail Flow of the GRUM's Unicast and Multicast Communication Models

GRUM API Module consist of two communication models; multicast and unicast communication models. **Figure 2** is the multicast communication model and its process flows. MBS Sending Data Model applies both GRUM's communication models to broadcast data to all the receivers. In **Figure 2**, before a sender is able to start broadcasting the messages to receivers, the sender has to inform the (GRUM's) *Coordinator* regarding the sending action (step 1). Once the *Coordinator* receives the sender's request, the request will be processed. As the request is going through the processing stage, the *Coordinator* replies to the sender that the current request is being received and is currently being processed (step 2). As the request is being processed in the processing stage, the coordinator will inform the sender that the current request is received and is currently being processed. Meanwhile, the *Coordinator* will validate the request (step 3 and step 4) and notify the sender if the sending action is approved. A ***BatchID*** will be dispatched if the approval is granted (step 5). When the sender obtains an ***Invoke*** message from the *Coordinator*, a ***Confirmation*** message for the sending action will be sent to the *Coordinator* (step 6) and the sender will start the broadcasting messages to all the receivers (step 7). When the sender obtained an invoke message from the coordinator, a conformation message of the sending action will be sent to the coordinator in return. After then, the sender will only start to broadcast the message to all the receivers.

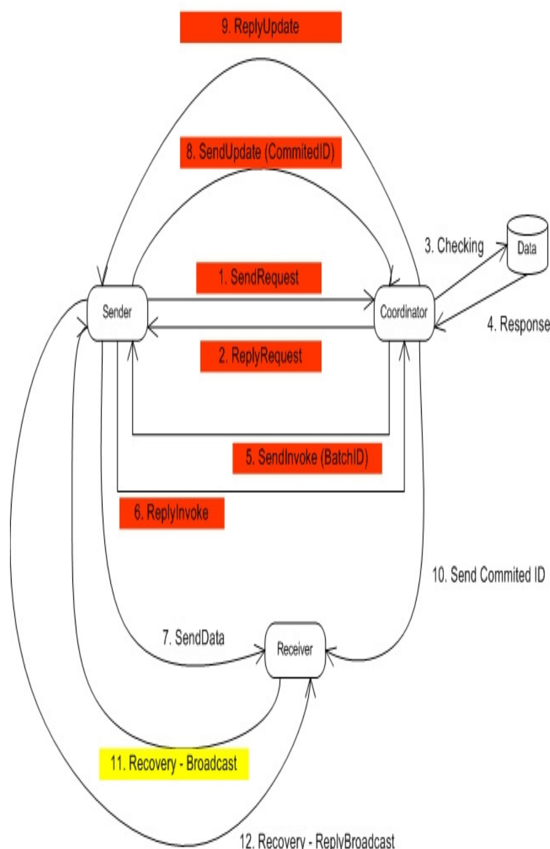

Figure 2: Multicast Communication Model

A ***BatchID*** will be assigned to every message sent and the ***BatchID*** value will increase by one for every sent message. As soon as the broadcasting

data completed, the sender will inform the *Coordinator* that the broadcasting event has been finished with a ***CommitedID*** (step 8). The ***CommitedID*** is the last ***BatchID*** assigned to the last message. It is an indication to all the member of the group so that each recipient is aware of the expected number of messages to be received. As long as the ***ReplyUpdate*** message reach at the *Coordinator*, the *Coordinator* will process the message and respond by notifying the sender that the current request is undergoing the processing stage (step 9). The *Coordinator* will then announce the ***CommitedID*** to all the receivers to publish the last message counter of the sender (step 10). All the receivers will be able to identify any lost messages based on the latest ***CommitedID*** broadcasted by the *Coordinator*. If there are any lost messages, the receivers will then ask for the *Messages Recovery* process to be activated for a particular sender using **Source-based Recovery** algorithm [2] (step 11). The sender will then broadcast the missing messages to the receivers which requested for the recovery process (step 12).

In **Figure 3**, the GRUM Unicast Communication Model and the process flows are illustrated. Before a sender begins sending the messages to a receiver, the sender has to request the permission from the *Coordinator* (step 1). The *Coordinator* will immediately notify the sender that the request is being received and processed (step 2). The *Coordinator* will validate the request (step 3 and step 4) and acknowledge the sender if the sending action is approved. If request is approved, a ***UniBatchID*** (step 5) will be given and sent to the sender. When the sender obtains the ***Invoke*** message from the *Coordinator*, a ***Confirmation*** message for the sending action will be replied to the *Coordinator* (step 6) and then the sender will only start the broadcasting services to the receiver (step 7). For every message sent, a ***UniBatchID*** will be assigned to the message and the ***UniBatchID*** value will be increase for every sent messages. When the sending process has completed, the sender will inform the *Coordinator* that the broadcasting event has been finished with ***UniCommitedID*** (step 8).

The ***CommitedUniBatchID*** is the last ***BatchID*** assigned to the last message of the sending activity. The *Coordinator* will notify the sender that the current request is being received and is under processing (step 9). The *Coordinator* will announce the ***CommitedUniBatchID*** to the receiver to indicate the latest message broadcasted by the sender and its last counter value (step 10). Once the information is received by receiver, an acknowledgement will be sent back to the *Coordinator* (step 11). The receiver will identify any lost messages based on the latest ***CommitedUniBatchID***. If there are any lost messages, the receivers will then request for a *Messages Recovery* service from the sender (step 12). The sender will again broadcast the missing messages to the receivers (step 13).

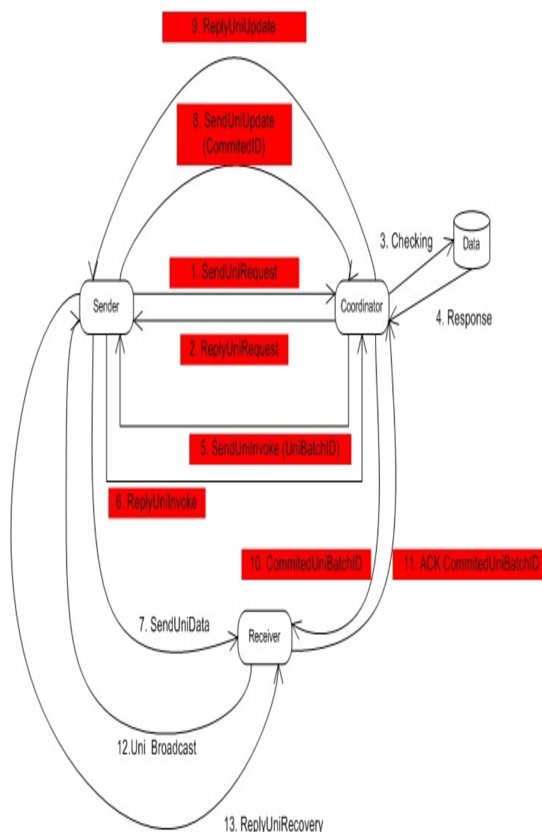

Figure 3: Unicast Communication Model

4. The Proposed MBS Design and Architecture

The proposed Multimedia Broadcasting System (MBS) utilizes a modular architecture. The proposed MBS architecture consists of few components (**Figure 4**). Fundamentally, the MBS system is organized into few modules; they are **Process**, **GRUM**, and **Service** modules. **Service** module composes of ***DB Factory***, ***Wrapper Factory***, ***Unicast*** and ***Multicast ReceivedEvent Handler***. ***DB Factory*** provides services for the execution of the sender, receiver and coordinator with the underneath object-oriented database (namely, db4o). In other words, ***DB Factory*** handles all the transaction issues such as insert, select, update and delete operations. ***Wrapper Factory*** is responsible for breaking up a file into multiple

fragments whereby all the size of each fragment is according to predefined block size. In addition, each fragment also encapsulates with extra information such as block identity, unique identifier, message type, command type and so on. The **Process** module is designed to handle the *Sending* and *Receiving* activities of data among the sender, receiver and coordinator. It is sub-divided into *Sending* and *Receiving* processes. *Sending* is responsible for the distribution of the commands and media files to all receivers, while *Receiving* is used to accept the incoming messages from the sender and performing the *joining* task once the files completed the receiving tasks. To enable *Sending* or *Receiving* processes, a request for service message to the GRUM API module is required before the process is executed.

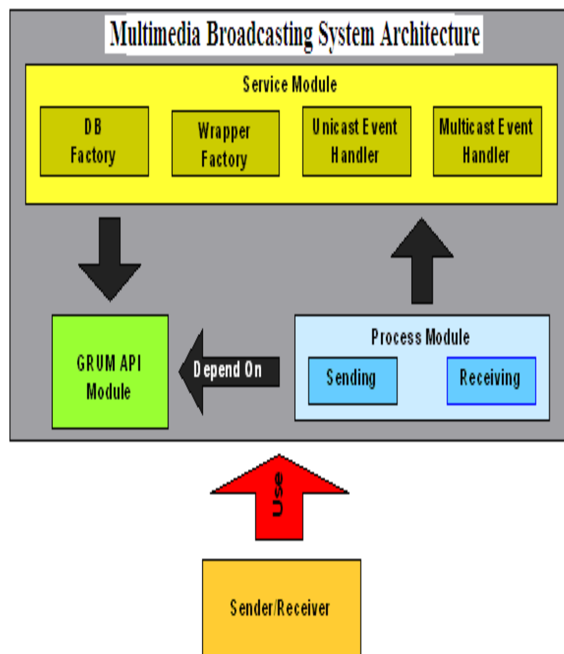

Figure 4: Multimedia Broadcasting System Architecture.

Unicast and ***Multicast ReceivedEvent Handler*** in charge of the reactions of the receivers towards the incoming data from the sender. For ***Unicast ReceivedEvent*** and ***Multicast ReceivedEvent handler***, both classes store the incoming messages into temporary object-oriented database storage for subsequent processing job. The implementation of the ***Unicast Event Handler*** and ***Multicast Event Handler*** is depended on the GRUM API to trigger all the required activities.

5. Design Strategies of MBS

MBS is a distributed multimedia player that was built using GRUM in order to have a reliable broadcast of media data to all receivers in remote locations on a not-so-reliable copper-based network infrastructure. It is a system that ensures the media data broadcasted by a sender are received completely and entirely by all the receivers within the same group.

5.1 The Flow and Detail Design of MBS

In the design of MBS data transferring process, there are several types of message in the communication activities. These message types are ***Data***, ***Command***, ***Information***, ***Construction*** and ***File***. ***Data*** are messages that contain data block that is part of the media file, message type, block identity and a unique identifier. The unique identifier is a random unique identifier generated through java utility facilities. Generally, every single media file that is broadcasted by the sender will be split into few blocks whereby the size of each block is according to a predefined size. Every single block will have a block identity which indicates the block arrangement sequence. For example, data block with block identity "2" specifies the second part of the media file. Once the receiver gathered all the blocks, receiver will construct the file back based on the block identity in ascending order. ***Command*** message is used to notify the multimedia player to play or stop the media files. ***Information*** message is the message that describes the media details such as the file name, file size, and total block. ***Construction*** message is used to construct the original file by combining the content within all the blocks in its original sequence. A ***File*** message indicates the media files that are successfully received by the receiver and keep track of the file's current location in a particular computer.

5.2 The Design of MBS Sending and Receiving Strategies Utilizing GRUM as the Communication Engine

MBS is designed to broadcast media files in various sizes. It also ensures that all these files are received by the intended receivers without being affected by the network infrastructure and reliability. The unpredictable Internet connection in some countries such as Malaysia, will not affect the broadcasting services and corrupt these files.

In **Figure 5**, the MBS enhances the GRUM's communication model between a sender, receivers

and coordinator by proposing additional features into it. Before the sender and receiver involve in the media files broadcasting, both of them have to register them to the GRUM coordinator. The GRUM coordinator is the centralized information repository for publishing the registered senders and receivers in a group. GRUM coordinator also checks registered senders and receivers whether they are active or inactive state by enquiring their current status. If any senders or receivers did not respond by sending a feedback to the GRUM coordinator after certain

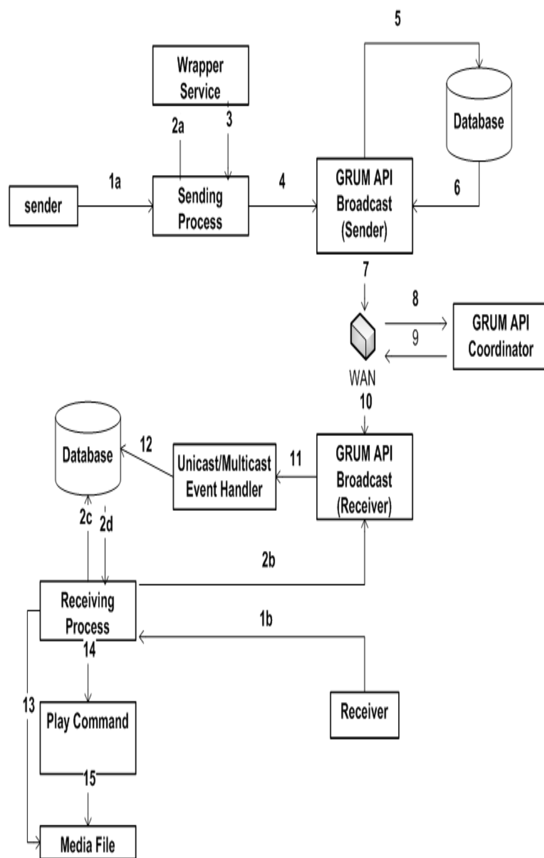

Figure 5: MBS Communication Model.

time interval, these senders or receivers are considered to be inactive and will be dropped out from its information repository or yellow page by the GRUM coordinator. Both senders and receivers must obtain the GRUM coordinator service agreement before they can proceed to other processes. The GRUM API **Broadcast** [1] is the communication layer between the sender, receiver and coordinator that provide the sending or receiving information services among each other. Wrapper Service has two responsibilities, splitting the big files into certain block sizes and encapsulate those data block before these block data are broadcasted. In addition, information such as block identity, unique identifier, and message type are appended to the message block. The Unicast and Multicast Event Handler are invoked when there are incoming messages from the senders. These messages will be acted upon by the appropriate actions or interfaces based on the receiving messages' detail.

5.2.1 Sending Data Model

Basically, sending data model involving steps from 1 to 10 (1a, 2a, 3, 4, 5, 6, 7, 8, 9 and 10) as shown in **Figure 5**. In the initial step, preliminary task likes identifying the receivers, select media files that need to be broadcast and determine the broadcast mode are the inputs need to provide by the sender before proceed to next step. In the following step, the MBS system begins to verify and validate those media files by compiling a media fragments or files broadcasting list. For each media file, it consists of file name, file size, total number of data block and a unique identifier which will be broadcasted to all receivers as an ***information message***.

Next, the ***Wrapper Service*** will split the media file to certain block size and assign each block with a block identity which will be used for joining process at receiver site. The output from the wrapper service is a data that had been encapsulated with information like message type, block identity and unique identifier. Thus, in this context, it is called ***data message***. This is followed by the entire data block will be broadcasted by using GRUM API Broadcast service to distribute those data to all receivers. However, before the data is being sent through the Internet, GRUM API Broadcast service will temporary store these data blocks into an object-oriented database as a queue. Before the GRUM API Broadcast service is able to begin the sending service, request for broadcasting permission from GRUM coordinator is required. Once the request is approved, the GRUM API Broadcast service will start transmitting the data based on the content and configurations in the queue to all relevant receivers.

5.2.2 Receiving Data Model

The receiving data model can be divided into pre-receiving data model and post-receiving data model. Pre-receiving data model is where receivers join the broadcast channel earlier than the senders while post-receiving data model is where receivers join the broadcast channel after the senders. In post receiving model, if the sender has broadcast part of the media file, the GRUM API broadcast service at receiver will continue receiving the data broadcast

by sender and establish a recovery mechanism to the sender for retrieving back the former data block.
Steps from 1b, 2b, 2c and 2d in **Figure 5** are the initial stage for the receiver to establish the communication with GRUM coordinator before the receiving process begins. The receiver will establish a communication session with GRUM coordinator for joining the group and ready to accept any broadcast signal from the sender. For pre-receiving data model, the GRUM ***Coordinator*** will create a new group based on the pioneer request. For post-receiving model, the group creation is based on the sender. From steps 11, 12, 13, 14, and 15, shows the receiving process at the receiver site. The incoming messages consist of two types, data message or playing message. Data message is an encapsulated data which contain the media file block, playing command is a command use for playing or stopping the current playing file.

For all the incoming data received by the GRUM API Broadcast service, the data will forward to Unicast/Multicast Event Handler for further processing. However, to avoid conducting a heavy processing at event handler, all the incoming message will temporary store into a database where by the message will be process later by another worker. The worker, which is a receiving process, will ensure all the data message is completely receive before the file reconstruction occurs. A ***Construction*** message will be created and store into the database if the transmitting completed. The receiving process will assign a new sub process taking responsible for processing those messages in the database base on the information provide in the ***Information*** message. As to reconstruct the origin file, the content inside the encapsulated data will be retrieved according to the block identity in sequence, extracting out and serialize to the targeted location for saving. As long as the construction process is finish, that particular message in the database will be modify to ***File*** message as a record to inform the player how many files are constructed and to be played.

6. Analysis of the GRUM'S Communication Models and the Performance of MBS

To successfully and reliably broadcast large media files to all receivers is the most important goal that GRUM and MBS must fulfill. GRUM Communication Models is selected as the communication protocol between the sender and receiver because GRUM guarantees the reliable multicast and unicast services. However, currently GRUM does not perform very well when trying to send large media files in a low bandwidth public switching network environment. In order to overcome this weakness, MBS has introduced a **Split and Join Later** algorithm to enable the very large media files to be sent in the low bandwidth public switching network environment. Before the sender broadcasts a large media file, the file will be split into few smaller parts and before sending through using GRUM's communication model. This splitting strategy will alleviate the sender resources usage in terms of memory and processing in broadcasting the file to all receivers. Meanwhile, the joining process will take place once the parts are reaching at receiver side. Since every part consist of a block identity, the joining process will construct back together by putting the received part to the right place in a file according to the block identity. Even if the parts with block identity arrives out of sequence, the joining process still can proceed without any problems.

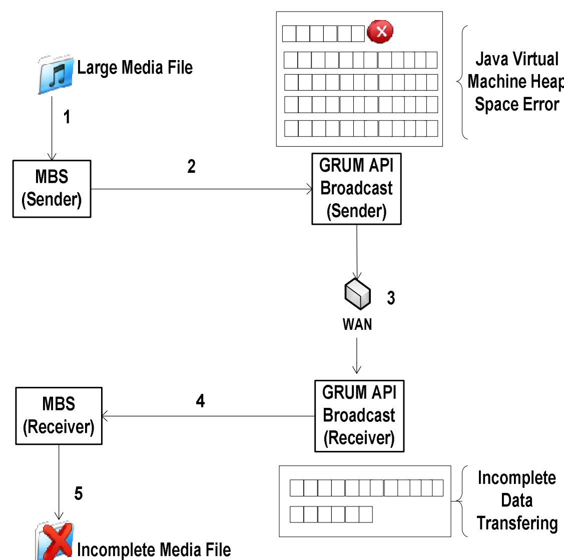

Figure 6: MBS without Split and Join implementation.

In **Figure 6**, the MBS was tested without the **Split and Join** enhancement. Steps 1 to 5 in **Figure 6** shows the activities flow of MBS broadcasting a large media file (any media files with more than 50 Meg) from sender to receiver. When the sender begins sending a large media file, the MBS performs without problem. However, as the heap space was consumed after some time intervals, the MBS starts to slow down and finally crashed. In the diagram, notice that as the GRUM API at the sender side hits the Java Virtual Machine (JVM) heap

space limit crashes when sending a large media file in its original architecture. This scenario demonstrated that GRUM's architecture required MBS to have large amount of heap resources from JVM to temporarily store transmitting data while the sender is broadcasting media files to all the receivers. As the result, the JVM is always unable to have sufficient free heap space or main memory for the large media files processing, hence, bringing down the MBS.

In **Figure 7**, the diagram shows that the MBS was enhanced with the **Split and Join** implementation. When the sender begins to send large media files, the **Split** module will split the media file by breaking the original file into few smaller parts and pass through GRUM API's **Broadcast** module for transmission. Notice that the GRUM API working smoothly without facing JVM heap space limit error. This is because the GRUM API only handles smaller files which require much lesser heap resources for the processing and broadcasting of files. As soon as the fragmented media pieces reached the receiver, the data will be stored in a temporary space for the processing of the **Join** module. Within the **Join** module, it reforms the fragmented media files into its original media file. The **Join** module will place every smaller and fragmented part to the correct 'block' in its original file structure by following the numbered block sequences.

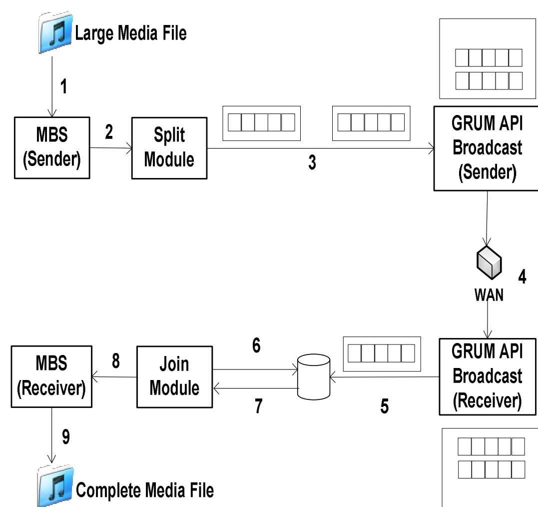

Figure 7: MBS with Split and Join implementation.

7. Testing, Evaluation, and Conclusion

In short, the improvement embedded into MBS to complement the GRUM API has considerably help when large media files are broadcasted to multiple receivers. The following table is statistics for a small samples collected in our latest run tests.

Media file size	With MBS's enhancement	Without MBS's enhancement
1-5Meg	OK	OK
5-10Meg	OK	OK
10-50Meg	OK	Memory break
50-100Meg	OK	Memory break

Table 2.0: Results comparing with and without MBS's enhancement.

The **Table 2.0** above showing the results of a system with and without implements the MBS. The purpose of this analysis is to show the differentiation between a system with and without implement the MBS. From the table 1, the system with MBS implementation can work smoothly without facing memory leak problems when the media file size is increasing. However, for the system without implement the MBS, the system always face memory breakdown issues when the media size growth up from 10 Megabytes and onwards. Thus, the system without implement the MBS cannot support for large media file transfers and unable to complete the data transferring process. Although the system with implement MBS in **Table 2.0** has shown that with the MBS's enhancement the sending for very large is still working very well even though it is slow. Therefore, the future research should venture into improving the GRUM's architecture especially the communication models of the system to use **Split and Join Later** algorithm.

8. References

[1] **TongMing, Lim, TechMin, Chin, ChiaChing, Teh and KhongMun, Chong**, "*A Guaranteed Reliable Asynchronous Unicast and Multicast Engine for Retail Chained-Stores*" (2008), submitted to PACIS2008 (*paper under reviewed*)

[2] **Danyang Zhang, Sibabrata Ray, Rajgopal Kannan, S.Sitharama Iyengar** (2003)**,** *A Recovery Algorithm for Reliable Multicasting in Reliable Networks*, Proceedings of the 2003 International Conference on Parallel Processing (ICPP'03)

[3] **Qixiang Sun, Daniel C. Sturman** (2000), *A Gossip-based Reliable Multicast for Large-scale High-througput Application*, IEEE

[4] **Christian Maihöfer, Kurt Rothermel** (2000), *Building Multicast Acknowledgment Trees, University of Stuttgart*, Institute of Parallel and Distributed High-Performance Systems (IPVR)

[5] **Christian Maihofe**r (2000), *Scalable and Reliable Multicast ACK Tree Construction with the Token Repository Service*, Institute of Parallel and Distributed High-Performance Systems (IPVR)

Copyright © 2008 by the International Business Information Management Association (IBIMA). All rights reserved. Authors retain copyright for their manuscripts and provide this journal with a publication permission agreement as a part of IBIMA copyright agreement. IBIMA may not necessarily agree with the content of the manuscript. The content and proofreading of this manuscript as well as and any errors are the sole responsibility of its author(s). No part or all of this work should be copied or reproduced in digital, hard, or any other format for commercial use without written permission. To purchase reprints of this article please e-mail: admin@ibima.org.