What is Wrong With the Transmission?

— A Comprehensive Study on Message Passing Related Bugs

Mingxing Zhang Yongwei Wu Kang Chen Weimin Zheng
Tsinghua National Laboratory for Information Science and Technology (TNLIST)
Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China
Research Institute of Tsinghua University in Shenzhen, Shenzhen 518057, China
Technology Innovation Center at Yinzhou,
Yangtze Delta Region Institute of Tsinghua University, Ningbo 315000, Zhejiang, China
Email: zhangmx12@mails.tsinghua.edu.cn {wuyw,chenkang,zwm-dcs}@tsinghua.edu.cn

***Abstract*—Along with the prevalence of distributed systems, more and more applications require the ability of reliably transferring messages across a network. However, passing messages in a convenient and dependable way is both difficult and error-prone. Thus the existing messaging products usually suffer from numerous software bugs. And these bugs are particularly difficult to be diagnosed or avoided. Therefore, in order to improve the methods for handling them, we need a better understanding of their characteristics.**

This paper provides the first (to the best of our knowledge) comprehensive characteristic study on message passing related bugs (*MP-bugs*). We have carefully examined the pattern, manifestation, fixing and other characteristics of 349 randomly selected real world MP-bugs from 3 representative open-source applications (OpenMPI, ZeroMQ, and ActiveMQ). Surprisingly, we found that nearly 60% of the non-latent MP-bugs can be categorised into two simple patterns: the message level bugs and the connection level bugs, which implies a promising perspective of detecting/tolerating tools for MP-bugs. Apart from this finding, our study have also uncovered many new (and sometimes surprising) insights of the message passing systems' developing process. The results should be useful for the design of corresponding bug detecting, exposing and tolerating tools.

Keywords*—*network; message passing; bug characteristics

I. Introduction

A. Motivation

With the increasing complexity of modern software system, many applications include components that stretch across some kind of network, either a LAN or the Internet. Thus application developers often end up doing some message passing routines. However, different from "merely transferring a few bytes between two nodes", "sending messages in a convenient and dependable way" is much more difficult for several reasons: *i)* the portability between different architectures — It is a common requirement that the messaging layer should be portable between different architectures (and even different languages). This leads to an intricate packing/unpacking procedure; *ii)* the variousness of communication specifications — In order to facilitate the users, a reusable messaging layer usually implements various communication specifications, which makes it large and complex. For example, ZeroMQ [1], a popular socket library, contains 12 different socket types which can be connected in 22 ways. And, as pointed out by DeSouza et al. [2], the MPI standard includes a total of 70 ways to implement a single point-to-point communication (14 send calls and five receive calls that can be combined arbitrarily); *iii)* the unreliability of network — The underlying network of a message passing system may be unreliable, which means that the messaging layer should be able to survive a temporary network partition or even network unavailable. It needs to handle lost messages and at the same time eliminates duplicate messages, whose complexity increases dramatically with the increasing size of the network; and *iv)* the dynamic nodes — The pieces in a network can be sluggish sometime, and even go away temporarily. Some persistence and fallback mechanisms are needed to be implemented for providing reliable massage passing. As a result, the existing messaging products are suffered from numerous software bugs. Even worse, these massage passing related bugs (*MP-bugs*) are difficult to diagnose or avoid, because specific topology or network condition may be required to reproduce them. There are even some MP-bugs that are nondeterministic, which means these bugs depend not only on inputs and execution environments, but also on thread interleaving and other timing-related events to manifest.

To address the above challenges, it will require efforts from multiple related directions, including the bug detecting, exposing, tolerating and fixing techniques. All of these directions have made some progress over the past years, but still remain many unsolved issues:

- How can the MP-bugs be properly categorized? A proper categorization of bugs can provide useful guidelines and motivations for the future researches. It can be used to answer many meaningful questions: Is there any pattern that many MP-bugs share? Is there any type of bugs that has not been addressed yet by the existing works? Is a custom method for some specific types of bugs meaningful? As an illustration, since most of the previous MP-bug detection researches [3], [4], [5], [6] are focused on MPI, which are constrained in not only applications but also communication patterns, a corresponding taxonomy are needed to answer the question: how important are the other patterns (e.g. publish-subscribe)?
- What is the result of real world MP-bugs? Apart from understanding the MP-bugs' causes, investigating the effects of MP-bugs is also important for a different set of reasons. Analyzing the effects can serve as a guide for further development not only of tools and methodologies that detect, but also of tools and methods designed to

tolerate and recover from the faults and errors caused by such bugs. To give a simple example, it is important to understand how often the MP-bugs cause corrupted messages, in order to gauge the effectiveness of using a separate dynamic verification tool and a corresponding resending system to automatically tolerate them.

- What is the manifestation condition of real world MP-bugs? The major challenge of software testing is the exponential input and interleaving space. In order to achieve a complete testing coverage of the programs, the testing cases need to cover every possible interleaving for each input test case, which is infeasible in practice. To address this challenge, an open question in testing is as follows: can we selectively test a small portion of the whole possible space and still expose most of the bugs? Thus the developing of testing techniques requires a good understanding of the MP-bugs' manifestation conditions. That is, we need to know what conditions are needed to reliably trigger a MP-bug. For example, how many brokers, and how many clients are usually involved in a real world MP-bug's manifestation?
- What is needed to reproduce a MP-bug? There is a close correlation between the reproducibility of a bug and the complexity of analyzing or fixing it. Thus understanding this question can help design more effective unit test and tracing technologies.
- How helpful are existing tools in diagnosing and fixing the real world MP-bugs detected by them? For example, some MP-bug detection tools can remind the programmers that some messages are not received by the corresponding peer. Such information can be very helpful when using a simple point-to-point communication pattern. But the communication can be much more complex (e.g., with multiple brokers or many layers of communication) thus more detailed information may be required in practice. More generally, we want to know that what information is needed while the programmers are fixing a real world MP-bug.

Answering the above questions will significantly benefit from a better understanding of the real world MP-bugs. However, although many empirical studies on general program bug characteristics (not specific to MP-bugs) have been done in the past [7], [8], [9], [10], and their findings have provided useful guidelines for the future researches, few studies have been conducted on real world MP-bugs. Previously, researchers have only conducted some preliminary works on the misuses of MPI [11], [12]. But they are constrained in a specific software, several simple communication patterns, and with no brokers or failover abilities.

B. Contributions

This work provides the first (to the best of our knowledge) comprehensive real world MP-bug characteristic study. Specifically, we examine the bug patterns, bug consequences, manifestation conditions, fixing complexity and other characteristics of real world MP-bugs. Our study is based on 349 randomly selected real world MP-bugs, collected from 3 large and mature open-source applications: OpenMPI, ZeroMQ, and ActiveMQ, which representing different levels of reliabilities. For each bug, we have examined every piece of information related to it, including the programmers' explanations in the bug report, forum discussions, source code patches, and the bug-triggering test cases. This information together provides us a relatively thorough understanding of the bugs.

Our study reveals many interesting findings, which provide useful guidelines for the future developing of MP-bug detecting, exposing, tolerating and fixing techniques. As an illustration, we found that nearly 60% of the non-latent MP-bugs (even the semantic bugs) can be categorized into two simple patterns: the message level bugs and the connection level bugs. This finding shows that even the semantic bugs in MP-bugs, which is often caused by application-specific reasons, shares some common patterns. And it is quite important for the tool developers who want to automatically detect or tolerate MP-bugs. We summarize our main findings and their implications in Table I. They and some more findings are elaborated in the following sections.

While we believe that the applications and bugs we examined well represent a large body of MP-bugs, we do not intend to draw any general conclusions about all message passing systems. In particular, we should note that all of the characteristics and findings obtained in this study are associated with the three examined applications and the programming languages these applications use. Therefore, the results should be taken with the specific applications and our evaluation methodology in mind (see Section II-C for our discussion about threats to validity).

The remainder of this paper is organized as follows. In Section II we describe our methodology. Then we present our findings on MP-bugs' pattern, manifestation and fixing in Section III, Section IV and Section V respectively. Other observations, such as the prevalent multi-layer architecture in message passing systems, will be briefly discussed in Section VI. And finally, Section VII discusses related works and Section VIII concludes.

II. Methodology

A. Application selection

Three open source applications are used in our study: OpenMPI, ZeroMQ, and ActiveMQ, in which OpenMPI and ActiveMQ are the de facto implementation of the MPI [13] and the AMQP [14] standard respectively and ZeroMQ is a high-performance asynchronous messaging library aimed at use in scalable distributed or concurrent applications. They are all mature and large applications that are popularly used in industry and possess well-maintained bug report databases.

More importantly, each of them represents a specific level of reliability: *i)* in OpenMPI, the library does not handle dynamic nodes at all (neither reconnect nor resend); *ii)* in ZeroMQ, the tool will maintain the connection automatically, but the messages are not durable[1] (reconnect but not resend); and *iii)* in ActiveMQ, it intends to provide a fully reliable message passing layer (both reconnect and resend). These three applications also have a good coverage on other directions: with/without broker, various communication patterns, different programming languages, etc.

[1]A "durable message" is a message that will be held on by some kind of reliable broker network if the corresponding receiver is temporarily unavailable.

Findings on Bug Patterns	**Implications**
(1) Semantic bug is the dominant root cause (264 out of 349) of the examined MP-bugs.	More efforts should be put into automatically detecting, exposing and tolerating semantic bugs.
(2) Nearly 60% (168 out of 289) of the non-latent MP-bugs are covered by two simple patterns: the *message level bugs* and the *connection level bugs*.	Tools that intend to detect, expose or tolerate message passing related bugs can focus on these two major bug categories, which are still very meaningfully.
(3) Only about 20% of the MP-bugs will immediately bring down or stall the system from making progress.	Proper asynchronous bug tolerating tools can handle a majority of MP-bugs.
(4) Communication patterns that are not covered by MPI are also prevalent in MP-bugs	New tools are needed to handle the other communication patterns related MP-bugs, which are not addressed by the existing works.
Findings on Manifestation Conditions	**Implications**
(5) The manifestation of most (312 out of 340) examined MP-bugs involves only a few (< 6) nodes.	Most of the MP-bugs can be detected by monitoring a relative small network, which reduces testing complexity without losing bug exposing capability much.
(6) Almost all the examined MP-bugs from ActiveMQ involve no more than 2 brokers.	2 brokers are enough for the testing of many sophisticated reliability features, such as durable messages, failover broker, etc.
(7) Nearly 40% of the MP-bugs in ActiveMQ are related to failover, which is far more than the ratio of corresponding code regions.	MP-bug testing tools could pay more attentions to this kind of bugs.
Findings on Bug Fixing	**Implications**
(8) About 40% of MP-bugs are not deterministic, which is much more than the result of previous studies on general bugs.	Bug diagnosis tools will need to incorporate new techniques (such as time-stamping inputs or controlling thread scheduling) in order to reproduce failures due to these bugs via input replaying.
(9) The time needed for fixing a MP-bug is several months on average.	This fact boosts the need of better bug tolerating techniques.
(10) The patches for MP-bugs are usually small; about 60% of the patches contain less than 20 lines of code and most of the patches only affect 1 file.	The hardness of fixing a MP-bug mainly concentrate on diagnosing why it will happen, not the following fixing step.
(11) The first patch of 79 (out of 349) MP-bugs we checked is buggy or inadequate.	Programmers need help to improve the quality of their patches; for example, the patch verification tools.

TABLE I: Main findings of this study and their implications.

B. Bug selection

The bug report databases of the selected applications contain a very large number of bugs. Thus we automatically filtered bugs that are not likely to be relevant by performing a search query on the bug report database. We searched the bug report databases for bugs that contained keywords commonly associated with MP-bugs, for example, 'message(s)', 'lost', 'duplicate', '(un)pack', 'order', etc. In addition to this we only searched for bugs whose status was closed, fixed, or resolved (i.e., bugs that are no longer under analysis by the developers), because the other bug reports that are marked as unclosed (or incomplete, etc.) are not likely to have enough information for our study. From the thousands of bug reports that contain at least one keyword from the above keyword set, we randomly sampled a subset (about 200) of the bugs for each application and manually analyzed them.

Phase \ App.	OpenMPI	ZeroMQ	ActiveMQ
Total closed bugs	955	204	2,165
Analyzed bugs	81	109	159

TABLE II: Bug counts of different applications.

The manual inspection revealed that some of the bugs that matched the search query are not message passing related and so we also excluded them. In addition, we also excluded bugs for which the corresponding bug report does not contain enough information to analyze them. After filtering, we obtained a final set of 349 analyzed MP-bugs, a number that is close (or even superior) to the number of bugs analyzed in many previous studies on other kinds of bugs [9], [15], [16]. Table II shows the bug counts of different applications.

C. Limitations

Our methodology's limitations should be considered when using or interpreting our results. Like other empirical studies, our results are limited by the kinds of applications and software bugs we used. While we believe that the applications and bugs we examined well represent a large body of MP-bugs, we do not intend to draw any general conclusions about all the massage passing systems. In particular, we should note that all of the characteristics and findings obtained in this study are associated with the three examined applications and the programming languages these applications use. Therefore, the results should be taken with the specific applications and our evaluation methodology in mind.

Moreover, we focus on the bugs that are found in the implementation of messaging systems temselves. Thus our study may not reflect the characteristics of other types of bugs, such as the misuse of massage passing systems.

In terms of our examination methodology, we manually analyzed the bug reports of the sampled MP-bugs. We have examined every piece of information related to each examined bug, including programmers' explanations, forum discussions, source code patches, and the bug-triggering test cases. In

Dimension	Category	Description
Root Cause	memory	Bugs caused by improper handling of memory objects.
	concurrency	Bugs that only happen in multi-threading (or multi-processes) environment.
	semantic	Inconsistent with the original design requirements or the programmers' intention.
Impact	message-level	Bugs that impact the delivering of one or several messages.
	connection-level	Bugs that are related to the connection between two nodes.
	node-level	Bugs that result in the termination of the complete system or some components of the system or causes extensive corruption of the data.
	latent	Bugs that will not affect the system's correctness immediately.
Communication Pattern	send-receive	Delivering messages between two pairs.
	collective	Communication that involves all the processes in a logical group of nodes.
	produce-consume	Each produced message will be consumed by one and only one consumer.
	publish-subscribe	All the messages published by the publishers will be received by all the subscribers.
	general	Bugs that do not specific to one communication pattern (e.g., bugs that related all kinds of connections).

TABLE III: Categories of the three dimensions. *Some categories and definitions are borrowed from BugBench [17].*

addition, we are also familiar with the examined applications, since we have used them in many of our previous projects.

III. Bug pattern study

To build a more reliable massage passing system, it is important to understand the type of bugs that are most prevalent and the typical patterns across MP-bugs. Since different types of bugs require different approaches to detect and tolerate, these fine-grained bug patterns provide useful information to developers and tool builders alike.

In order to provide guidelines for future research on MP-bugs, in this section, we focus on three particular dimensions: **Root Cause**, **Impact** and **Communication Pattern**. Specifically, *i)* along the cause dimension, we classify the MP-bugs into three disjoint categories based on their root causes; *ii)* for the impact dimension, we analyzed the MP-bugs with respect to their consequences that are exposed to the users, and divided these impacts into four categories and 13 sub-categories; and *iii)* when it comes to the communication patterns, we have totally observed five different modes. All the related definitions are given in Table III and Table IV.

A. Partition by root causes

We first present the analysis results based on the bugs' root causes. As mentioned before, we classify the MP-bugs into three disjoint categories: *memory*, *concurrency*, and *semantic*. Figure 1, which shows the total number of each type of bugs across different applications, summarizes our investigation results.

Finding (1): Semantic bug is the dominant root cause (264 out of 349) of the examined MP-bugs.
Implication: More efforts should be put into automatically detecting, exposing and tolerating semantic bugs.

From the figure, we can observe that semantic bug is the dominant root cause of MP-bugs. It counts to 67% - 84% of the examined MP-bugs. One possible reason of this fact is that most semantic bugs are application specific. A programmer can easily introduce semantic bugs due to a lack of thorough understanding of the system, its requirements or its specifications. Thus it is much harder to automatically detect them. In contrast, the causes of memory and concurrency bugs are general for any applications, which means that the developers can use general tools for exposing them. As an illustration, Valgrind [18] is a popular instrumentation framework for building dynamic analysis tools. There are many mature Valgrind tools [19], [20] that can automatically detect a lot of memory management and concurrency bugs. Moreover, there also exist techniques [21], [22], [23] that intend to tolerate these bugs on-the-fly in the production runs. They can help the system to avoid failures, even when the bug is already triggered.

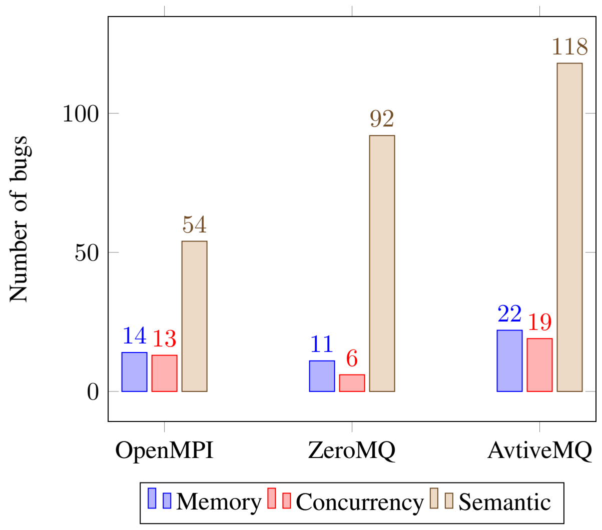

Fig. 1: Distribution of root causes.

Although the result is a little bit lower than the 81.1% - 86.7% reported in a previous general bug characteristics study [24] (bugs excluding memory bugs and concurrency bugs); it is still very significant. Therefore, there is no doubt that more efforts should be put into automatically detecting, exposing and tolerating semantic bugs.

B. Partition by impacts

As we have discussed above, a large portion of MP-bugs are caused by application specific reasons, which hinders the mining of programmers' intensions. As a result, it is hard to automatically detect or tolerate these bugs. However, for the message passing related bugs there exist two kinds of

Category	Sub-category	Description	Abbr.
Message Level Bugs	Corrupted message	The message received is not the same as the sent message.	Cor.
	Duplicate message	The message is received twice or more.	Dup.
	Lost message	One or more messages are deemed delivered but not received by the corresponding receivers (i.e., the sending procedure is accomplished successfully, but the messages are lost).	Los
	Unclaimed message	Received unsent message.	Unc.
Connection Level Bugs	Failed topology maintaining	The message passing system fails to discover/remove (i.e., setup/close a connection with) an active/closed node, or a newly added node fails to add itself to the network.	Top.
	Failed automatic reconnection	The reconnection procedure caused by temporary network partition or node restarting is failed.	Rec.
	Out of order	The user assumes a specific order of message with the connection, but this intension is not guaranteed.	Ord.
Node Level Bugs	Crash	Several nodes or the whole system become unusable	Crash
	Error	Operation failure or unexpected error code returned.	Error
	Hang	One or more nodes of the message passing system become making no progress (e.g., infinite loop, deadlock, live lock).	Hang
Latent Bugs	Leak	Some resources are not freed after usage (e.g., memory leak, file descriptor leak).	Leak
	Performance loss	The execution take more resources (e.g., time, CPU, bandwidth) than expected.	Per.
	Uncategorized latent bug	The other consequence, such as incorrect statistic information.	Unl.

TABLE IV: Bug Consequence Classification. *This table shows the definitions of various MP-bug consequences.*

intentions that are both important and prevalent: the message related intensions and the connection related intensions.

Message related intensions: the user of a message passing system usually wants his/her message to be intactly sent to the correct receiver once and only once, but this intention may not be guaranteed. The message can be lost because of the unreliability of the network or software errors, at the same time the corresponding resending procedure may cause duplicate messages. And it is also possible that the message is corrupted during the packing/unpacking procedure or some receivers received a message that is not sent by any sender.

Connection related intensions: In order to deliver messages, the message passing system needs to maintain the topology of all the active nodes (senders, brokers, receivers, etc.), which is difficult since the network is unreliable and the nodes may be dynamically added/removed. As a result, there are many bugs related to this connection maintaining procedure, such as unable to reconnect two nodes or fail to discover a newly added node.

It will be very helpful to gauge the portion taken by these two kinds of intentions among MP-bugs. In order to answer this question, we study the impacts of MP-bugs, and categorize them following the definitions listed in Table IV. Among the four categories, the *Message Level Bugs* and the *Connection Level Bugs* correspond to the aforementioned message related intensions and connection related intensions respectively. In the other two categories, the *Latent Bugs* are less important, since *i)* mature tools [25] can be used to detect memory leakages, which is the dominate part of *Leak*; *ii) Performance loss* is not that serious since it does not impact the system's correctness; and *iii)* the remaining uncategorized latent bugs are usually incorrect statistic information, which is not harmful if not used. That is, only the *Node Level Bugs'* corresponding intensions remain unclear.

Table V presents the statistic results. Note that the sum of all occurrences is larger than the total number of bugs because some bugs fit into more than one category.

Type \ App.		OpenMPI		ZeroMQ		ActiveMQ	
Message Level Bugs	Cor.	27	23	26	12	45	13
	Dup.		1		0		7
	Los.		1		12		23
	Unc.		2		2		2
Connection Level Bugs	Top.	6	6	25	15	39	21
	Rec.		0		10		16
	Ord.		0		0		2
Node Level Bugs	Crash	39	20	45	22	37	10
	Error		9		15		9
	Hang		10		8		18
Latent Bugs	Leak	11	3	14	6	40	14
	Per.		5		6		20
	Unl.		3		2		6

TABLE V: Consequence of MP-bugs.

Finding (2): Nearly 60% (168 out of 289) of the non-latent MP-bugs are covered by two simple patterns: message level bugs and connection level bugs.
Implication: Tools that intend to detect, expose or tolerate message passing related bugs can focus on these two major bug categories, which are still very meaningfully.

Apart from the latent bugs that will not lead to a failure immediately (and even not harmful in sometimes), nearly 60% of the remaining MP-bugs are message/connection related. This is also true for semantic bugs, 143 out of 220 non-latent semantic bugs belong to these two categories. As we have mentioned before, these bugs are much easier to be automatically detected than the general semantic bugs, because the programmers' corresponding intensions are more apparent. According to our investigation, we have summarized a total of seven different popular intentions, which correspond to the seven sub-categories under the *Message Level Bugs* and the *Connection Level Bugs*. And we think specific tools can be built to detect or tolerate each of them.

For example, FlowChecker [26] is a low-overhead method for detecting MP-bugs in MPI libraries. It instruments the binary code of both MPI applications and MPI libraries which logs MPI function calls (e.g., MPI_Send and MPI_Recv) at the application level and data movement operations at the library level into trace files. Then, by investigating the MPI calls in the trace files, FlowChecker tracks the corresponding message flows by following the relevant data movement operations starting from the sending buffers. If the message data are not correctly delivered to the receiving buffers as indicated by the MPI calls, FlowChecker reports the bug and provides diagnostic information, such as faulty MPI functions or incorrect data movements, to help pinpoint the root causes. To put it simply, FlowChecker can be used to detect all the four sub-categories of message level bugs in MPI. And we can easily anticipate that similar method can be used for the other applications (apart from MPI).

As for the connection level bugs, a standby ping/pong system can be used as a reference for the verification (just like the checksum technique used in detecting message level bugs). Increasing the system's reliability by implementing multiple functionally equivalent programs, which is based on the classic idea of N-version programming [27], has already been successfully used in many other fields. As an illustration, EnvyFS [28] implements a thin VFS-like layer near the top of several replicates file-system (e.g., ext3, ReiserFS, JFS) and uses majority-consensus to operate correctly despite the sometimes faulty behavior of an underlying commodity child file system. However, we do not (as far as we know) find a mature tool that can handle connection level bugs, although they account for a great portion when reliability is needed.

Finding (3): Only about 20% of the MP-bugs will immediately bring down or stall the system from making progress.
Implication: Proper asynchronous bug tolerating tools can handle a majority of MP-bugs.

Among all the 13 sub-categories of the MP-bugs, only $Crash$ and $Hang$ will immediately bring down or stall the system from making progress. This give space to asynchronous bug tolerating tools. That is, the user can run the message passing system in tandem with a dynamic verification tool (e.g., FlowChecker, standby ping/pong system). Once observing a bug's happening, another remedial system can try to do some recovery (e.g., resending, reconnecting, restarting). At least it can explicitly signal the message passing system, and hope it will be handled by the client application appropriately.

C. Partition by communication patterns

Finally, we study the distribution of communication patterns among MP-bugs.

Finding (4): Communication patterns that are not covered by MPI are also prevalent in MP-bugs.
Implication: New tools are needed to handle the other communication patterns related MP-bugs, which are not addressed by the existing works.

As we have mentioned before, as fast as we know, most of the previous MP-bug detection researches [2], [**?**], [5] are focused on MPI, which are constrained in not only applications but also communication patterns. However, the result shows that other patterns are also important. Moreover, these uncovered patterns (i.e., produce-consume, publish-subscribe) can not be converted to point-to-point communication easily; because more information, such as the current network's topology, is needed to verify their correctness. That is, although

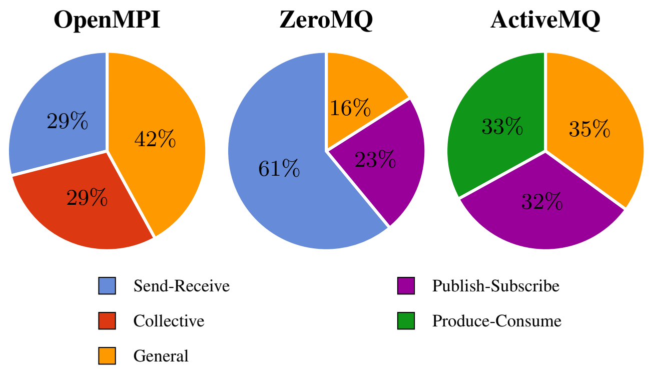

Fig. 2: Distribution of communication patterns.

important and prevalent, these other patterns have not been well studied by previous researches. Thus many bugs will be missed by the existing MP-bug detectors, which mainly focus on send-receive bugs or collective bugs that can be observed in MPI.

IV. Bug Manifestation Study

The manifestation condition of an MP-bug is usually a specific order among a set of message or system events. In this section, we study the characteristics of real world MP-bug's manifestation. We will discuss the learned guidelines for MP-bug testing and detecting based on our observations.

A. How many nodes are involved?

Finding (5): The manifestation of most (312 out of 340) examined MP-bugs involves only a few (less than 6) nodes.
Implication: Most of the MP-bugs can be detected by monitoring a relative small network, which reduces testing complexity without losing bug exposing capability much.

Finding (5) tells us that even though the examined message passing systems usually deployed in a large network (hundreds of nodes), in most cases, only a small number (less than 6) of nodes are involved in the manifestation of a MP-bug.

The underlying reason for this is that most messages do not closely interact with many nodes, and most communication and collaboration is conducted between small group of nodes. As a result, the manifestation conditions of most MP-bugs do not involve many nodes. Moreover, 234 out of the 340 successfully analyzed MP-bugs can be manifested in the simplest network topology (2 for MPI/ZeroMQ, 3 for ActiveMQ).

We should note that this finding is not opposite to the common observation that MP-bugs are sometimes easier to manifest at a heavy-workload or in a large network. The heavy-workload will increase the resource competition and context

Application	$\geq$ 6 nodes	5 nodes	4 nodes	$\leq$ 3 nodes
OpenMPI	9	0	8	55
ZeroMQ	5	0	5	99
ActiveMQ	14	7	38	100

TABLE VI: The number of nodes involved in MP-bugs. *Some bugs are omitted if not enough information is given.*

switch intensity. It therefore increases the possibility of hitting certain orders that can trigger the bug. The manifestation condition still involves just a few nodes.

Finding (6): Almost all the examined MP-bugs from ActiveMQ involve no more than 2 brokers.
Implication: 2 brokers are enough for the testing of many sophisticated reliability features, such as durable messages, failover broker, etc.

We further study the number of brokers involved in the MP-bugs from ActiveMQ. The results are given in Figure 3. It shows that almost all the examined MP-bugs from ActiveMQ involve no more than 2 brokers, which means that 2 brokers are enough for the testing of many sophisticated reliability features, such as durable messages, failover broker, etc.

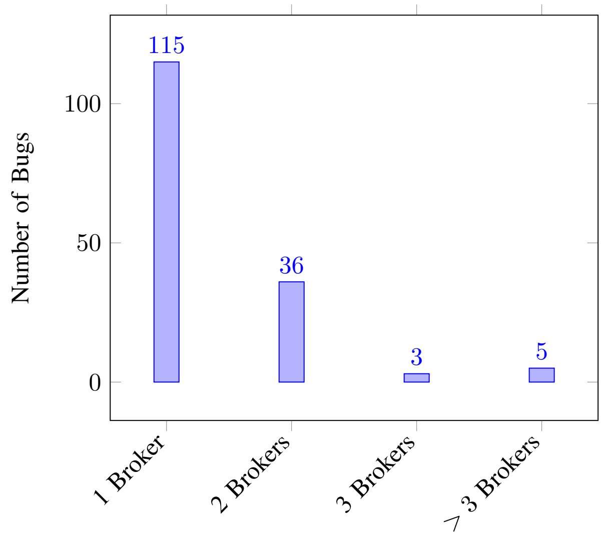

Fig. 3: The number of brokers involved in MP-bugs from ActiveMQ.

Overall, our finding implies that the testing of MP-bugs can focus on small networks. Such testing technique can prevent the testing complexity from increasing exponentially with the number of nodes. At the meantime, few MP-bugs would be missed. Such implication is a necessary condition of practical model checking for message passing systems.

B. How many MP-bugs are related to failover?

How hard is the achieving of reliable message passing? To answer this question, we examine the number of bugs that are related to the failover procedure (e.g., reconnecting to a restarted node). According to our investigation, many bugs we found arose not in common-case code paths but rather in the more unusual failover-handling cases. This type of bugs is critical for the messaging layer's robustness, since they usually hinder the fulfillment of programmers' intention on reliabilities. In the rest of this section, we will quantify bug occurrences on failover-handling paths; Tables VII (a), (b) and (c) present our investigation results.

OpenMPI	ZeroMQ	ActiveMQ
3 (4%)	14 (13%)	59 (37%)

(a) By Application.

Memory	Concurrency	Semantic
7 (15%)	11 (29%)	58 (22%)

(b) By Root Cause.

Message-level	Connection-level	Node-level	Latent
12 (12%)	34 (49%)	20 (17%)	10 (15%)

(c) By Impact.

TABLE VII: Failover Related Bugs. *This table shows the number and percentage of the bugs related to failover in MP-bugs.*

Application	Deterministic	Input Timing Dependent	Non-deterministic
OpenMPI	57	1	23
ZeroMQ	77	12	20
ActiveMQ	88	28	43

TABLE VIII: Symptom Reproducibility Characteristics.

Finding (7): Nearly 40% of the MP-bugs in ActiveMQ are related to failover, which is far more than the ratio of corresponding code regions.
Implication: MP-bug testing tools could pay more attentions to this kind of bugs.

As we can see from the first table, only a few bugs are related to failover in OpenMPI, since it only provides a less-used checkpoint system. But the importance of failover-related bugs increases with the increasing requirements of reliability; at last, nearly 40% of the MP-bugs in ActiveMQ are related to failover, which is far more than the ratio of corresponding code regions.

When broken down by bug type and impact in the second and third table, we can see that roughly 20% of semantic bugs occur on the failover paths and in node level bugs. We think this can be another breakthrough for inferring the programmers' intentions of semantic bugs.

V. Bug fixing study

A. Bug reproducibility

Reproducing bug symptoms is a prerequisite for performing automatic bug diagnosis. In this section we subclass the

MP-bugs by their reproducibility. That is, can the bug be reproduced on demand? And do bugs have characteristics that ease or hinder automatic bug diagnosis? Table VIII summarizes our results; the bugs are classified as either deterministic, input timing dependent, or non-deterministic.

A failure due to a software bug is observed to be deterministic if the fault triggers the same symptom each time the application is run with the same set of input requests in the same order, on a fixed architecture/OS platform and a fixed server configuration. Otherwise, the bug is non-deterministic. We also single out a special case of non-deterministic software bugs named input timing dependent. A bug is deemed as input timing dependent if the timing of the input requests in addition to their order determines whether a symptom is triggered and if so which symptom. In contrast, if the occurrence of the symptom depends upon timing issues that are beyond the client's control (e.g., the thread scheduling for concurrency bugs); we still classify such failures as non-deterministic, not as input timing dependent.

Note that our study is conservative with respect to the definition of non-determinism. More specifically, we classify a bug as non-deterministic if, according to the bug report, the symptom(s) could not be reproduced consistently for any reason (e.g., the same inputs may not be available or parts of the environment might have changed). It is possible that such bugs are deterministic, but we conservatively assume that they are not. Also, our definition of determinism is overly restrictive. We believe that many of the deterministic bugs in our study are actually deterministic across different environments and configurations as, in many cases, failures are reproduced by developers on different systems from the one where the bugs are first detected. But because bug reports often lack sufficient information about the bug's behavior across different environments, we chose to define deterministic to mean reproducibility in a fixed environment. In some bug reports, the failure could not be reproduced or was very difficult to reproduce (as it occurred infrequently). We conservatively classified such bugs as non-deterministic.

Finding (8): About 40% of MP-bugs are not deterministic, which is much more than the result of previous studies on general bugs.
Implication: Bug diagnosis tools will need to incorporate new techniques (such as time-stamping inputs or controlling thread scheduling) in order to reproduce failures due to these bugs via input replaying.

These properties are useful, as it will determine whether the diagnosis tools can reliably reproduce the failure symptoms. And fortunately, the results show that about 60% of the bugs demonstrate deterministic behavior, which means they can be reproduced by given the same set of input requests in the same order.

However the remaining 40% of MP-bugs, which are not deterministic, are also non-ignorable. This result is much more than previous Sahoo's results (17%) on general bugs [9]. One possible reason is that many MP-bugs are related to networks and multiple nodes, which are much complex. Additionally, the existing bug detection tools can effectively reduce the diagnosis and resolution time of deterministic bugs.

As a result, since the majority of MP-bugs are deterministic, bug diagnosis tools should be able to reproduce them by replaying inputs. But there still exist a large body of MP-bugs that are non-deterministic. Thus bug diagnosis tools will need to incorporate new techniques (such as time-stamping inputs or controlling thread scheduling) in order to reproduce failures due to these bugs via input replaying.

B. Bug fixing complexity

Finally, we compared MP-bugs from different categories with respect to their complexity of fixing them, according to the bug report fields that specify these properties. For measuring the complexity of fixing bugs we used three metrics that can be extracted from the bug reports: time to fix the bug, number of files involved in the patches, and line of code changed. Although none of these metrics is perfect, in combination they help us estimate the complexity of fixing these bugs.

Category	Time	File	LoC
Memory	102/17	3/1	115/8
Concurrency	207/59	4/2	70/22
Semantic	140/20	2/1	68/14
Total	142/21	2/1	75/15

TABLE IX: Complexity of fixing MP-bugs. *For each class of bugs we present the average/median for each of the three metrics: time to fix the bug in days, number of files in the patches and line of code changed.*

We present a comparison of the three complexity metrics in Table IX. Since some of these fields contain significant outliers, in addition to presenting the average for all three metrics we also present the median.

Finding (9): The time needed for fixing a MP-bug is several months on average.
Implication: This fact boosts the need of better bug tolerating techniques.

Our analysis of the fixing complexity revealed that the bug fixing is hard in term of consumed time. It usually takes several months on average to fix a bug. This fact boost the requirement of bug tolerating techniques, since even the bug is detected the user still need to wait an appreciable period of time before using a correct version of code. And the techniques that can automatically generate a temporary patch (just like Loom [29] for concurrency bugs) are also useful.

Finding (10): The patches for MP-bugs are usually small; about 60% of the patches contain less than 20 lines of code and most of the patches only affect 1 file.
Implication: The hardness of fixing a MP-bug mainly concentrate on diagnosing why it will happen, not the following fixing step.

However, the bugs' fixing complexity is not that apparent in terms of patch size. As we can see from the table, most of the patches are small. And, according to our investigation, about 60% of the patches contain less than 20 lines of code and most of the patches only affect 1 file. We can infer from this fact that the hardness of fixing a bug is concentrate in diagnosing why it will happen, not the following fixing step.

C. Mistakes during bug fixing

Another indicator for the hardness of fixing a MP-bug is that many patches released by programmers are still buggy. In order to investigate the nature of buggy patches, we count the number of patches applied to each bug, and the relation between bugs.

Finding (11): The first patch of 79 (out of 349) MP-bugs we checked is buggy or inadequate.
Implication: Programmers need help to improve the quality of their patches; for example, the patch verification tools.

Our study finds that the first patch of 79 (out of 349) MP-bugs we checked is buggy or inadequate; 32 of these bugs are explicitly reopened by the developers, which means that the programmers once incorrectly determine the bug to be fixed after their testing. Among these distinct buggy patches, some of them only decrease the occurrence probability of the original MP-bugs, but fail to fix the bug completely. And, more important, 19 of them introduce new bugs. This shows that programmers need help to improve the quality of their patches; for example, the patch verification tools.

VI. Other characteristics

Many bugs rely on specific system architecture: In our study, we find many bugs are relied on one specific system architecture or even the cooperation of machines with several specific architectures. The majority of these bugs are related to the packing/unpacking procedure of the messages. But some of them are much trickier. For example, bug $OpenMPI\#213$ is caused by the different alignment strategy of different architectures. Specifically, the reporter consistently got a segmentation fault caused by SIGBUS when running the following line:

$$\begin{aligned} hdr->hdr\_match.hdr\_ctx = \\ sendreq->req\_send.req\_base \\ .req\_comm->c\_contextid; \end{aligned}$$

After investigation, the developer finds that the hdr field may be oddly aligned in SPARC, which caused the code to fire a SIGBUS because hdr_ctx requires a 2 byte alignment. Thus the programmers fix the bug by manually padding the variable.

Bug severity: Bug severity is about the risk a bug poses if it gets out into the wild. Usually the severity of a bug can belong to the following types [30]: *i) Critical:* The failed function is unusable and there is no acceptable alternative method to achieve the required results. *ii) Major:* The failed function is unusable but there exists an acceptable alternative method to achieve the required results. *iii) Minor:* The defect does not result in termination, but causes the system to produce incorrect, incomplete or inconsistent results. *iv) Trivial:* The defect does not result in termination, does not damage the usability of the system and the desired results can be easily obtained by working around the defects. We study the MP-bugs with respect to their severity, according to the bug report fields that specify these properties. We found that only 70 out of 349 MP-bugs are labeled as critical, which means that an acceptable alternative method can be found for most of the bugs. It raises the possibility of implementing bug tolerating tools.

Some MP-bugs cross many layers: Nowadays, the software become more and more complex. Some times the path of a message-sending request will traverse many layers. And the request also appears differently at each layer. For example, a message-sending request in JBossMQ will incur a corresponding message sending in the underlining ActiveMQ. This, in turn, results in Ethernet packets across the network, and finally leads to another message-receiving request at the corresponding peer. We also find bugs that cross many layers within one software. As an illustration, a message in ZeroMQ can be split into many sub-messages, and each sub-message is transmitted independently. Thus the sending of the whole message and the sending of each sub-message can be determined as two layers. This prevalent layering increases the hardness of bug diagnosis. Tools that can automatically pin-point to the right layer will be very helpful.

VII. Related work

Bug characteristic studies Given the importance of software reliability and the prevalence of bugs, many studies on bug characteristics have previously been done. These works have studied many aspects of various kinds of bugs, including their patterns, impacts, reproducibility, and fixes [15], [7], [8]. And many of them provide precious information to help improve software reliability from different aspects, such as bug detecting, fault tolerance, failure recovery, and testing, etc [16], [9].
But unfortunately, few studies have been conducted on real world MP-bug characteristics. Previously, researchers have conducted some preliminary works on the misuses of MPI [11], [12]. But they are constrained in a specific software, several simple communication patterns, and with no brokers or failover abilities. In contrast, our work provides a much more comprehensive study on general MP-bugs.

MPI bug detection The Message Passing Interface library is one of the most popular message passing layer in practice, and is being actively developed and supported through several implementations designed to run on a plethora of architectural platforms. Thus many conventional debugging tools have been designed for MPI programs, such as the Intel Message Checker [2]. There also exist works for detecting the bugs in the developing of MPI libraries. For example, FlowChecker [26] can extract program intentions of message passing, and to check whether these intentions are fulfilled correctly by the underlying MPI libraries, i.e., whether messages are delivered correctly from specified sources to specified destinations. If not, it reports the bugs and provides diagnostic information. Another line of tools such as ScalaTrace [31] and MPIWiz [32] record MPI calls into a trace file and use this information to deterministically replay the program. Our work is complementary to these works, since it has the potential to guide and motivate the development of these kinds of techniques and approaches.

Verification solutions for message passing systems In addition to the aforementioned testing approaches, researchers have also explored formal verification and model checking methods for detecting bugs in the message passing systems. Spin [33], a tool for analyzing the logical consistency of concurrent systems, provides built-in support for a number of message passing features. It has been used to verify message passing systems by many researchers [3], [4]. Some recent works [5], [6] take an approach that integrates the best features of testing tools (ability to run directly on user applications) and model checking (coverage guarantees) to verify MPI applications. They run the MPI program under the control of a verification scheduler, guarantee to detect all potential matches for non-deterministic (wildcard) receives, and explore each of these matches in different runs of the program. Thus, they exhaustively explore and ensure full coverage of non-determinism. Our study on MP-bug manifestation conditions provides guidelines for the future researches of this field, which can help understand the trade-off between testing complexity and bug exposing capability and help design better coverage criteria.

VIII. Conclusion and future work

This paper provides a comprehensive study of the massage passing related bugs. We examined their pattern, manifestation, fixing and other characteristics. Our study is based on 349 real world MP-bugs, randomly collected from 3 representative opensource programs: OpenMPI, ZeroMQ, and ActiveMQ. The result of our study includes many interesting findings and implications, which can benefit future researches on MP-bug detecting, exposing and tolerating in various aspects. For example, future work can design new bug detection tools to address the message level bugs and the connection level bugs; can focus on testing relative small networks, etc. In particular, we intend to develop tools that can automatically tolerate MP-bugs.

IX. Acknowledgments

This Work is supported by National High-Tech R&D (863) Program of China (2012AA012600), Natural Science Foundation of China (61433008, 61373145, 61170210, U1435216), Chinese Special Project of Science and Technology (2013zx01039-002-002).

References

[1] "http://zeromq.org/."

[2] J. DeSouza, B. Kuhn, B. R. de Supinski, V. Samofalov, S. Zheltov, and S. Bratanov, "Automated, scalable debugging of MPI programs with intel&reg; message checker," ser. SE-HPCS '05.

[3] O. S. Matlin, E. L. Lusk, and W. McCune, "SPINning parallel systems software," in *Proceedings of the 9th International SPIN Workshop on Model Checking of Software*. London, UK, UK: Springer-Verlag, 2002, pp. 213–220. [Online]. Available: http://dl.acm.org/citation.cfm?id=645881.672236

[4] S. Siegel and G. Avrunin, "Verification of MPI-based software for scientific computation," in *Model Checking Software*, ser. Lecture Notes in Computer Science, S. Graf and L. Mounier, Eds. Springer Berlin Heidelberg, 2004, vol. 2989, pp. 286–303. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-24732-6_20

[5] S. Vakkalanka, G. Gopalakrishnan, and R. M. Kirby, "Dynamic verification of MPI programs with reductions in presence of split operations and relaxed orderings," ser. CAV '08.

[6] A. Vo, S. Vakkalanka, M. DeLisi, G. Gopalakrishnan, R. M. Kirby, and R. Thakur, "Formal verification of practical MPI programs," ser. PPoPP '09.

[7] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler, "An empirical study of operating systems errors," ser. SOSP '01.

[8] L. Lu, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and S. Lu, "A study of linux file system evolution," *Trans. Storage*, vol. 10, no. 1, pp. 3:1–3:32, Jan. 2014. [Online]. Available: http://doi.acm.org/10.1145/2560012

[9] S. K. Sahoo, J. Criswell, and V. Adve, "An empirical study of reported bugs in server software with implications for automated bug diagnosis," ser. ICSE '10.

[10] T. Zimmermann, N. Nagappan, P. Guo, and B. Murphy, "Characterizing and predicting which bugs get reopened," ser. ICSE '12.

[11] S. Sharma and G. Gopalakrishnan, "Efficient verification solutions for message passing systems," in *IPDPS Workshops*, 2011, pp. 2026–2029.

[12] S. F. Siegel and G. S. Avrunin, "Verification of MPI-based software for scientific computation," ser. SPIN '04.

[13] "http://www.mpi-forum.org/docs/docs.html."

[14] "http://www.amqp.org/."

[15] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from mistakes: A comprehensive study on real world concurrency bug characteristics," ser. ASPLOS '08.

[16] P. Fonseca, C. Li, V. Singhal, and R. Rodrigues, "A study of the internal and external effects of concurrency bugs," in *DSN'10*, 2010, pp. 221–230.

[17] S. Lu, Z. Li, F. Qin, L. Tan, P. Zhou, and Y. Zhou, "Bugbench: Benchmarks for evaluating bug detection tools," in *In Workshop on the Evaluation of Software Defect Detection Tools*, 2005.

[18] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," ser. PLDI '07.

[19] K. Serebryany and T. Iskhodzhanov, "Threadsanitizer: Data race detection in practice," ser. WBIA '09.

[20] N. Nethercote and J. Seward, "How to shadow every byte of memory used by a program," ser. VEE '07.

[21] M. Zhang, Y. Wu, S. Lu, S. Qi, J. Ren, and W. Zheng, "Ai: A lightweight system for tolerating concurrency bugs," ser. FSE 2014.

[22] J. Yu and S. Narayanasamy, "Tolerating concurrency bugs using transactions as lifeguards," ser. MICRO '43.

[23] J. Yu and S. Narayanasamy, "A case for an interleaving constrained shared-memory multi-processor," ser. ISCA '09.

[24] M. Sullivan and R. Chillarege, "A comparison of software defects in database management systems and operating systems," in *Fault-Tolerant Computing, 1992. FTCS-22. Digest of Papers., Twenty-Second International Symposium on*, 1992, pp. 475–484.

[25] M. Hauswirth and T. M. Chilimbi, "Low-overhead memory leak detection using adaptive statistical profiling," ser. ASPLOS '04.

[26] Z. Chen, Q. Gao, W. Zhang, and F. Qin, "Flowchecker: Detecting bugs in MPI libraries via message flow checking," ser. SC '10.

[27] A. Avizienis, "The methodology of n-version programming," *Software fault tolerance*, vol. 3, pp. 23–46, 1995.

[28] L. N. Bairavasundaram, S. Sundararaman, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Tolerating file-system mistakes with EnvyFS," ser. USENIX '09.

[29] J. Wu, H. Cui, and J. Yang, "Bypassing races in live applications with execution filters," ser. OSDI '10, 2010, pp. 1–13.

[30] "ISTQB Exam Certification. http://istqbexamcertification.com/."

[31] M. Noeth, P. Ratn, F. Mueller, M. Schulz, and B. R. de Supinski, "Scalatrace: Scalable compression and replay of communication traces for high-performance computing," *J. Parallel Distrib. Comput.*, vol. 69, no. 8, pp. 696–710, Aug. 2009. [Online]. Available: http://dx.doi.org/10.1016/j.jpdc.2008.09.001

[32] R. Xue, X. Liu, M. Wu, Z. Guo, W. Chen, W. Zheng, Z. Zhang, and G. Voelker, "MPIWiz: Subgroup reproducible replay of mpi applications," ser. PPoPP '09.

[33] G. Holzmann, "The model checker SPIN," *Software Engineering, IEEE Transactions on*, vol. 23, no. 5, pp. 279–295, May 1997.