Measuring Interprocess Communications in Distributed Systems

Xiaoqin Fu
Washington State University, Pullman, WA
xiaoqin.fu@wsu.edu

Haipeng Cai
Washington State University, Pullman, WA
haipeng.cai@wsu.edu

***Abstract*—Due to the increasing demands for computational scalability and performance, more distributed software systems are being developed than single-process programs. As an important step in software quality assurance, software measurement provides essential means and evidences in quality assessment hence incentives and guidance for quality improvement. However, despite the rich literature on software measurement in general, existing measures are mostly defined for single-process programs only or limited to conventional metrics. In this paper, we propose a novel set of metrics for common distributed systems, with a focus on their interprocess communications (IPC), a vital aspect of their run-time behaviors. We demonstrated the practicality of characterizing IPC dynamics and complexity via the proposed IPC metrics, by computing the measures against nine real-world distributed systems and their varied executions. To demonstrate the practical usefulness of IPC measurements, we extensively investigated how the proposed metrics may help understand and analyze various quality factors of distributed systems, ranging from maintainability and stability to security and performance, on the same nine distributed systems and their executions. We found that higher IPC coupling tended to be generally detrimental to most of the quality aspects while interprocess sharing of common functionalities should be promoted due to its understandability and security benefits.**

***Index Terms*—Distributed system, dynamic measurement, interprocess communication, coupling, quality factors**

I. Introduction

Measuring software systems in terms of properly chosen metrics is an integral step in software quality assurance [1]. Defining appropriate software metrics is essential for both software process quality and product quality, throughout the entire software development lifecycle [2]. Prior to the implementation phase, software metrics provide a means for specifying quality requirements with respect to relevant quality factors. After implementation, the metrics further serve as crucial guidance for evaluating the software product with respect to the specification of quality requirements. Software metrics also play a vital role in software project management as a whole (e.g., for cost and effort estimation) [3].

Two main classes of metrics can be used in software measurement: static and dynamic [4]. In comparison, static metrics are generally easier to compute relative to dynamic counterparts [5], [6]. Additionally, static metrics are not subject to limited code coverage or generalizability as are dynamic metrics. On the other hand, static metrics are not sufficient for measuring and interpreting dynamic behaviors of software, for which dynamic metrics offer much more precise indicators. In fact, concerning quality factors that are ultimately attested at runtime (e.g., performance [6], reliability [7], and testability [8]), dynamic metrics are much more preferable. Meanwhile, understanding software behaviors does not always need complete code coverage [9], thus the limited executions that dynamic metrics address does not necessarily constitute a constraint of dynamic measurement. Dynamic metrics cannot simply be (over-)approximated by corresponding static metrics either—in some cases, they are not even correlated [10].

Intuitively, distributed software systems could benefit from dynamic metrics for understanding and quality assessment/improvement as well. In fact, such benefits would be of peculiar significance for two reasons. First, understanding and quality assurance (e.g., testing and debugging) of distributed systems are especially difficult [11], due to their typically great size and complexity along with intrinsic concurrency and non-determinism [12]. Dynamic measurements would provide an avenue to overcoming such difficulties. Second, most real-world, industry-scale software systems are distributed by design, to accommodate increasing demands for performance and scalability. Addressing distributed software is thus of a high priority in the field of software quality assurance. In this context, it is crucial to develop and exercise well-informed measurements of distributed systems, especially those regarding the unique challenges (e.g., understanding their communication topology) posed by their distributed architecture.

Yet despite the rich literature on software metrics in general, the majority of existing metrics only concern centralized (mostly single-process) software [3], [5], [8], [13], [14]. A few works address distributed systems measurement, yet with a scope other than the software itself (e.g., monitoring system environments [15] or user dynamics [16]). In particular, a key facet of the run-time behaviors of distributed systems lies in those induced by interprocess communications (IPC), which has not been well addressed in software measurement [17].

In this paper, we develop a novel suite of IPC metrics aiming to measure the run-time communication structure, complexity, and reusability of common distributed systems [12], where constituent components[1] run concurrently on physically separated locations (nodes) while communicating and coordinating via message passing without a global clock (i.e., the timing mechanism). Specifically, we measure the *coupling* between concurrent, distributed processes at method, class, component/process, and whole-system levels, in terms of messaging and method-level dynamic dependencies across process boundaries. While coupling has been extensively studied among various types of software metrics [4], static or dynamic, it has not yet been addressed as regards the communication among processes during the execution of distributed systems.

[1]We define a component as the code entities that run in a separate process.

Measuring interprocess coupling in distributed systems, however, is challenging [18]. Coupling metrics are often defined on the basis of certain relationships (e.g., dependency and inheritance) [4]. However, deriving the interprocess dependencies, from which our metrics are computed, is not trivial in the context of distributed system executions. The main reason lies in the lack of global timing across the system together with the lack of explicit references/invocations across distributed components. To overcome this challenge, we leverage a framework for dynamic dependence analysis of distributed programs we recently developed [19]. Using this framework, we reason about interprocess dependencies through the happens-before relation between executing methods across processes, derived from a global partial ordering of method execution events. We further exploit the semantics of message passing to improve the precision of such derived dependencies, so as to enhance the validity of our IPC coupling metrics.

Using the proposed metrics, we measured the IPC traits of nine real-world distributed software projects, mostly enterprise-scale systems, in varied operation scenarios (with respect to different types of tests). To demonstrate the usefulness of the coupling measurement in aiding the analysis of distributed systems quality, we explored the relationships between coupling measures and six quality factors quantified through respective direct metrics, including *maintainability, correctness, stability, understandability, security, and performance*. We then conduct extensive statistical analyses to discover significant associations between our coupling metrics and the quality factors, via a non-parametric correlation analysis (with Spearman's method [20]). We showed, through empirically validated correlations, positive or negative, promising applications of our metrics in understanding the run-time behaviors of complicated distributed systems executions, and in studying various aspects of distributed software quality.

In summary, this paper makes the following contributions:

- We proposed a novel set of metrics for measuring IPC at varied (from method through system) levels in distributed systems executions, based on message-passing semantics and method-level dynamic dependencies.
- We applied the IPC metrics to nine real-world distributed systems, and hence characterized various aspects of run-time behaviors of these systems with respect to IPC.
- We extensively investigated the correlation between IPC measures and six quality factors of distributed software, and hence validated the usefulness of these IPC metrics.
- We made our study utilities and datasets publicly accessible [21] to facilitate future research on distributed systems measurement and quality assessment.

To the best of our knowledge, this is the first comprehensive study of dynamic coupling measurement in distributed systems in relation to various quality factors. Notably, since each process is the run-time instance of its corresponding component and components in distributed software are decoupled, by measuring the IPC coupling we essentially reveal the hidden/implicit coupling among distributed components. Beyond our technical approach and empirical results, we also distill major lessons learned from our study and provide recommendations for quality enhancement to distributed systems developers.

```
1  // The Server component
2  public class SC {
3      Datastore ds=null; Socket ssock=null;
4      void setup (int port) {ssock=new Socket(port); ds=dlDB(...);}
5      int serve() {long dataid=Long.valueOf(ssock.readLine());
6          return ssock.write(ds.retrieve(dataid));}
7      public static int main(String[] args) {S s=new S();
8          s.setup(Long.valueOf(args[1])); return s.serve();}}
9  // The Client component
10 public class CC1 {
11     public static int main(String[] args) {
12         CC2 conn=new CC2(); CC1 c=new CC1();
13         long dataid=Long.valueOf(args[1]);
14         System.out.println(c.lookup(conn,dataid)); return 0;}
15     String lookup(CC2 conn,long id) {return conn.getD(id);}}
16 public class CC2 {
17     Socket csock=connectServer(...);
18     String getD(long id) {csock.write(id); return csock.read();}}
```

Fig. 1. A distributed system as an illustrating/working example.

II. Motivating/Working Example

By design, a distributed system consists of multiple collaborating components each running in a process typically located at a separate computing node. Since these components interact primarily through IPC [22], understanding (measuring) IPC is essential for understanding (measuring) the behaviors of distributed systems. To illustrate, consider the simple example of Figure 1, where the system includes two components. The *Sever* component, implemented in class `SC`, provides the service for querying a database `ds` (line 4) according to the `dataid` sent by the client (lines 5–6). The *Client* component, implemented in classes `CC1` (for data querying) and `CC2` (for managing connections), prepares a query from user inputs (line 13) and then looks up the database server (line 14) through IPC (line 18). Apparently, IPC dominates the activities of this system, excluding which each component alone would be largely trivial. Thus, the complexity of this system's execution essentially lies in the complexity of its IPC.

Suppose the developer now wants to diagnose the communication security issues of this system reported by users who also provided the inputs that can reproduce the issues. A rewarding first step would be to understand how the communication currently works in terms of the IPC with respect to the user inputs. Measuring IPC would help in this scenario. Moreover, the IPC measurements may help the developer assess quality aspects other than security. For instance, if the IPC coupling is very high, intuitively the system might be difficult to update. Thus, its maintenance cost may be quite high also. Unfortunately, there is a lack of tool support for measuring distributed systems executions with respect to IPC, and it is unknown which quality factors might be analyzable through the IPC measures. In the rest of this paper, we address these questions by developing new IPC metrics and applying them to distributed software quality assessment.

Application scope. While our focus in this paper is on distributed systems, the proposed metrics apply to any systems that are similar to distributed systems in terms of the decoupled structure. For example, systems adopting microservice architectures or IPC-based inter-component communication [23] can also be measured using our IPC metrics.

III. Proposed IPC Metrics

This section presents our approach to IPC measurement. We first give the preliminaries underlying the definition of our IPC coupling metrics, summarizing the core rationales of our method-level dynamic dependence approximation. Then, we elaborate each of the metrics with justification and illustration, which measure *run-time message coupling (RMC)*, *run-time class coupling (RCC)*, *class central coupling (CCC)*, and *inter-process reuse (IPR)*. We motivate each metric by discussing its potential use for evaluating relevant quality factors.

A. Preliminaries

All of our proposed metrics are based on dynamic dependencies at method level across processes. Given a method ${P_i}^m$ in one process P_i, all methods in any another process P_j, $j \neq i$ in the systems execution that *depends* on ${P_i}^m$ form a set, referred to as the *dependence set* of ${P_i}^m$. Next, we outline below the two core steps of our approach for determining the dynamic dependence between two methods across processes. Further relevant details can be found in [24].

Step 1: partial ordering of method execution events. The basis of our dynamic method-level dependence approximation is happens-before relations between method entrance and exit events [25]. These events are partially ordered using the Lamport timestamps (LTS) algorithm [26], realized in dedicated runtime monitors for those method execution events. In addition, two types of communication events, message sending and message receiving, are monitored to update the per-process logic clocks in the LTS algorithm. Then, a method $m2$ is considered dependent on $m1$ if the first execution event of $m1$ happens before the last execution event of $m2$, according to the global partial ordering.

Step 2: pruning based on message-passing semantics For a more precise dynamic dependence approximation, we further leverage the semantics of message-passing events across processes. Suppose methods $m1$ and $m2$ are executed in two processes P_i and P_j respectively, and $m2$ is considered dependent on $m1$ according to the purely control-flow-based approximation described above (i.e., *Step 1*). From a data-flow perspective, $m1$ would not affect $m2$ if during the systems execution (1) P_i never sends any messages to P_j, or (2) the event that P_j receives the first message from P_i never happens before the last execution event of $m2$. The rationale is intuitive: for a method in one process to influence (i.e., causing dependence to) a method in another process, the two processes must have actually communicated (by passing messages from one to the other), and the possible dependence between the two methods again relies on their happens-before relation. Spurious dependencies as a result of *Step 1* are pruned using these two intuitive rules. Then, with this significantly more precise [24] dependence computation, we compute the dependence set underlying our IPC coupling metrics, as detailed next.

B. RMC (Runtime Message Coupling)

Definition/computation. We define interprocess message coupling at both process and system levels. First, given two different processes P_i and P_j, their message coupling $RMC(P_i, P_j)$ is intuitively defined as the number of messages sent from P_i to P_j. This process-level metric can be computed according to the communication events monitored in our framework (Section III-A). Then, the system-level RMC is defined as the average of such process-level measures overall all communicating pairs of processes as (hereafter, we suppose the system runs in N processes in total)

$$RMC = \frac{\sum_{j=1}^{N}\sum_{i=1}^{N} RMC(P_i, P_j)}{N(N-1)}, i \neq j, i, j \in [1, N] \quad (1)$$

Rationale/justification. At a high level, processes in distributed systems executions interact through passing messages. Thus, the value of the RMC metric indicates the extent of run-time interactivity among system components. A higher RMC implies greater reliance of a component on (i.e., sending more messages to) others in the execution considered. Since these components are distributed over separate locations, larger RMC values also indicate higher communication costs (e.g., for greater network bandwidth use). Finally, a higher RMC suggests greater effort for systems understanding (even at a very-high level) [27]. Thus, RMC can be used as an *understandability* and *cost-of-quality* [1] metric. In essence, RMC measures the implicit coupling between (statically) decoupled components because the two processes involved in the definition correspond to two components of the system.

Illustration. Consider the example system of Figure 1. Suppose during the system execution under analysis the client process (P_{client}) sends the server process (P_{server}) two messages—the first for authentication (line 12) and the second for querying (line 18), and the server then sends back to the client the querying result in three messages. Thus, $RMC(P_{server}, P_{client})$=3 and $RMC(P_{client}, P_{server})$=2. Since we have two processes (N=2), the system-level message coupling is RMC=(2+3)/(2(2-1))=2.5. For a second example, consider ZooKeeper [28], an enterprise-scale distributed system that consists of three major components each running in one or multiple processes. Further, consider its built-in integration test for an example execution, during which only one server and one client process are involved and they exchanged 13 messages during the test. Thus, both process-level RMC measures are 13 and the system-level RMC=(13+13)/(2(2-1))=13.

C. RCC (Runtime Class Coupling)

Definition/computation. We measure finer-grained interprocess coupling at class level, from which we define the class coupling between two processes hence further the system-level class coupling. Specifically, the coupling metric for two classes, P_i^A in process P_i and P_j^B in process P_j, is defined as the ratio of the total number of methods in P_j^B that are dependent on any method in P_i^A, to the total number of methods in any process other than P_j that are dependent on any method in P_i^A. Let $DS(m)$ denotes the dependence set of method m, the class-level RCC metric is defined as

$$RCC(P_i^A, P_j^B) = \frac{|\bigcup_{m \in P_i^A}\{f | f \in DS(m) \wedge \ f \in P_j^B\}|}{|\bigcup_{m \in P_i^A} DS(m)|} \quad (2)$$

Given a query m, the entire dependence set $DS(m)$ includes all methods that dynamically depend on m for the execution being analyzed [29], regardless of the dependant methods being executed in the same process as the query m (i.e., *local* process) or in other processes (i.e., *remote* processes). Accordingly, the dependant methods in the local and remote processes form the *local dependence set* and *remote dependence set*, respectively. The denominator in Equation 2 is the size of the union set of entire dependence sets of all methods in P_i^A, while the numerator is the size of the union set of remote dependence sets of those methods.

Next, the process-level RCC metric is defined as

$$RCC(P_i, P_j) = \sum_{A \in P_i} \sum_{B \in P_j} RCC(P_i^A, P_j^B), \qquad (3)$$
$$where \quad RCC(P_i^A, P_j^B) \neq \ 0$$

While the constraint $RCC(P_i^A, P_j^B){\neq}0$ is trivial for computation, it is not for the definition which only considers the pairs of classes that are actually coupled—a zero RCC would indicates they are not coupled.

Finally, the system-level RCC metric is defined as

$$RCC = \frac{\sum_{j=1}^{N} \sum_{i=1}^{N} RCC(P_i, P_j)}{N(N-1)}, i \neq \ j, i, j \in [1, N] \quad (4)$$

Rationale/justification. The rationale for RCC is that its value indicates how methods from a class in one process access methods from a class in another process. In contrast to RMC which is a *message coupling* metric, RCC measures functionality coupling. A higher RCC implies greater functional interdependency among system components in the execution considered. From a change management perspective, this higher RCC suggests that making changes for the associated system use case would be more difficult and costlier. Also, the denser interprocess dependence associated with a greater RCC makes it harder to test and debug the system with respect to the use case. Thus, RCC may inform about *maintainability* [1].

TABLE I
ILLUSTRATING DEPENDENCE SETS OF THE SYSTEM OF FIGURE 1

Method m	Dependence set $DS(m)$
`SC::main`	{`SC::main`, `SC::setup`, `SC::serve`, **CC1::main**, **CC1::lookup**, **CC2::getD**}
`SC::setup`	{`SC::setup`,`SC::serve`, **CC2::getD**}
`SC::serve`	{`SC::serve`, **CC2::getD**}
`CC1::main`	{`CC1::main`,`CC1::lookup`,`CC2::getD`, **SC::setup**,**SC::serve**}
`CC1::lookup`	{`CC1::lookup`,`CC2::getD`, **SC::serve**}
`CC2::getD`	{`CC2::getD`}

Illustration. For example, consider an operation profile of the system of Figure 1, in which the method-level dependencies[2] are listed in Table I. For each query (first column), the entire dependence set is given in the second column, including methods executed in remote processes as marked in boldface. Based on these local/remote dependencies, for class P_{server}^{SC}, the union set of all its methods' entire dependence sets is {`SC::main`, `SC::setup`, `SC::serve`, `C1::lookup`, `C2::getD`} while the union set of all the methods' remote dependence sets that belong to P_{client}^{CC1} is {`CC1::main`, `CC1::lookup`}. Thus, $RCC(P_{server}^{SC}$, $P_{client}^{CC1})$=2/5=0.4. Similarly, $RCC(P_{server}^{SC}$, $P_{client}^{CC2})$=1/5=0.2, $RCC(P_{client}^{CC1}$, $P_{server}^{SC})$=2/5=0.4, and $RCC(P_{client}^{CC2}$, $P_{server}^{SC})$=0/1=0. Then, the process-level RCC measures are $RCC(P_{server}$, $P_{client})$=0.4+0.2=0.6, $RCC(P_{client}$, $P_{server})$=0.4. Finally, the system-level RCC is computed as (0.6+0.4)/(2(2-1))=0.5.

[2]For the simplicity of our illustrations, we dismissed methods `dlDB` and `connectServer`, which are both *library* routines.

D. CCC (Class Central Coupling)

Definition/computation. On the basis of the RCC metric, we further measure the aggregate coupling as regards an individual class executed in a local process with respect to classes in all possible remote processes. Specifically, given a class P_i^A in process P_i, the CCC metric is defined as

$$CCC(P_i^A) = \sum_{j=1}^{N} \sum_{B \in P_j} RCC(P_i^A, P_j^B), \qquad (5)$$
$$where \quad RCC(P_i^A, P_j^B) \neq \ 0, i \neq \ j, i, j \in [1, N]$$

Again, the constraint $RCC(P_i^A, P_j^B){\neq}0$ is given for the rigor of the definition despite its triviality in the metric computation. The system-level CCC is then defined as the mean of class-level CCC measures over all classes executed (in any process).

Rationale/justification. Intuitively, the CCC metric of a class c characterizes the importance of c in terms of its influence (coupling strength) on all classes in remote processes by being coupled with them. If we construct a coupling graph where each node is a class and each edge represents RCC between two classes, the CCC metric is akin to the *centrality* metric in network measurement [30]. Our definition of CCC was originally motivated by the centrality metric indeed, which has been widely used in network analysis to identify the most important vertices in a graph (network). Thus, the CCC metric can be used to understand key classes in IPC, which could potentially inform about localizing quality issues (e.g., identifying faulty classes). Further, this metric may be utilized to assess functional *correctness* and other quality factors of the distributed system under measurement at class level.

Illustration. Let us consider the same example system (Figure 1) and execution scenario as used for illustrating the RCC metric. Given all those class-level RCC measures, we can readily compute the CCC measures for the three classes: $CCC(P_{server}^{SC})$=$RCC(P_{server}^{SC}$,$P_{client}^{CC1})$+$RCC(P_{server}^{SC}$,$P_{client}^{CC2})$ =0.4+0.2=0.6, $CCC(P_{client}^{CC1})$=$RCC(P_{client}^{CC1}$,$P_{server}^{SC})$=0.4, and $CCC(P_{client}^{CC2})$=0. The system-level CCC measure is thus computed as (0.6+0.4+0)/3=0.33.

E. IPR (InterProcess Reuse)

Definition/computation. Complementing the previous three forms of coupling metric, we further propose a metric of interprocess coupling at method level. And the system-level metric is derived from the method-level measures. Specifically, given a method P_i^m in any process P_i, let $LDS(P_i^m)$ and $RDS(P_i^m)$ denote the local dependence set (set of methods in P_i that depends on P_i^m) and remote dependence set (set of methods that depends on P_i^m but are in any process other than P_i) of P_i^m, respectively. Also, we denote as M the entire set of methods covered in the system execution under analysis—M is the union set of methods executed in any process during the execution: $M = \bigcup_{i \in [1,N]} \{P_i^m | P_i^m \in P_i\}$.

First, the method-level IPR metric with respect to an individual method P_i^m in a process P_i is defined as

$$IPR(P_i^m) = \frac{|LDS(P_i^m) \bigcap RDS(P_i^m)|}{|M|} \quad (6)$$

Then, the system-level IPR metric is defined as

$$IPR = \frac{\sum_{P_i^m \in M} IPR(P_i^m)}{|M|}, i \in [1, N] \quad (7)$$

Rationale/justification. The IPR metric measures functionality overlapping and code reuse at *method* level, across the processes in the system execution under analysis. The key rationale for this metric is that its value indicates the size of functionalities shared between system components. Real-world distributed systems are usually found to have common code modules used by two or more distributed components. As a typical example, the process-level common functionalities exercised reflect the fact that multiple components of the distributed system use the same third-party libraries. Therefore, IPR can serve as an intuitive metric measuring component-level code reusability [18]. On the other hand, despite the name of this metric suggesting code reuse, IPR is essentially still an (albeit derivative/variant) metric of interprocess coupling. Intuitively, a higher IPR metric value of a system indicates more dynamically coupled processes of the system.

Illustration. For the same example system and execution as used for illustrating other metrics and in reference to the dependence sets of Table I, the method-level IPR for any of the six methods covered by the execution is zero. Thus, the system-level IPR is also zero. (Note that the methods in boldface form the remote dependence set of each query.) For an alternative example, let us consider the library functions `dlDB` and `connectServer`. Now suppose these two functions both invoked another library function `libHttpConn` (for managing HTTP connections). Then, LDS(`SC::main`) and RDS(`SC::main`) would be {`SC::main`, `SC::setup`, `libHttpConn`, `SC::serve`} and {`CC1::main`, `CC1::lookup`, `libHttpConn`, `CC2::getD`}, respectively. Thus, IPR(`SC::main`) would be 1/7. Similarly, we may obtain IPR(`SC::setup`)=1/7, IPR(`CC1::main`)=1/7, and the IPR of other methods remain zero. Accordingly, the system-level would be (1/7+1/7+1/7)/7=3/49.

In sum, we defined four IPC metrics, covering four levels of granularity of measurement: method (IPR), class (RCC, CCC), component/process (RMC, RCC), and whole system (RMC, RCC, CCC, IPR). Some of these metrics (RMC, RCC) explicitly measure interprocess coupling, while others (CCC, IPR) are in essence derivatives/variants of coupling metrics. Note that IPR complements the other three metrics in that it measures the common dependencies of two processes while the other three measure mutual dependencies of one process on the other. It is also important to note that, for each of the proposed metrics, a metric value is not supposed to be examined on its own or in absolute terms. Instead, the metric value should be best interpreted in the context of comparable values obtained from a group of compatible measurement objects [2] (e.g., measurements with the same metrics against peer distributed system subjects). Next, we present our empirical experiments measuring real-world distributed systems and their executions using these IPC metrics. To make sure that each of the proposed metrics is appropriate for empirical studies, we have validated these metrics with respect to, and confirmed satisfaction of, their representation conditions [3].

TABLE II
STATISTICS OF EXPERIMENTAL SUBJECTS

Subject (version)	#SLOC	#Methods	Test type	#Cov.M.
OpenChord (1.0.5)	9,244	736	integration	382
Thrift (0.11.0)	14,510	1,941	integration	266
xSocket (2.8.15)	15,760	2,209	integration	466
ZooKeeper (3.4.11)	62,194	5,383	integration	979
			load	750
			system	797
Voldemort (1.9.6)	115,310	20,406	integration	715
Netty (4.1.19)	162,923	12,389	integration	1,325
Derby (13.1.1)	688,376	7,460	integration	291
Karaf (2.4.4)	60,978	1,012	integration	223
XNIO (2.0.0)	5,282	1,164	integration	113

IV. Experiment Methodology

The goal of our empirical studies is two-fold. First, we aim at characterizing the IPC in real-world distributed systems and their executions, in order to demonstrate the practicality of measuring distributed-system IPC using the proposed metrics. Second, we aim at investigating the implications of IPC characteristics (in terms of the IPC metrics) to the quality of distributed software in terms of various quality factors, in order to demonstrate the practical usefulness of measuring IPC for understanding, analyzing, and even predicting those quality factors. This section focuses on clarifying our experimental design, including our study subjects, datasets on the quality factors of interest, guiding research questions, and procedure.

A. Measured Systems and Executions

Table II lists the nine Java subjects used in our study, including the subject name and version (first column), subject size in terms of the number of non-comment non-blank code lines (second column) and the number of methods (third column). For generating the executions needed for computing our dynamic metrics, one or more types of test case (one test per type) were used used (the fourth column). The last column gives the number of methods covered by each test case.

Subject description. OpenChord is a distributed hash table used as a peer-to-peer network service [31]. Thrift is a scalable development framework for developing cross-language services [32]. xSocket is an NIO-based library for the development of high-performance network applications [33]. ZooKeeper [28], [34] is a widely used distributed coordination service for consistency and synchronization. Voldemort [35] is a distributed database (key-value store) used at LinkedIn. Netty is a non-blocking I/O event-driven framework used for rapid development of Java network protocol servers and clients [36]. Derby is an open source relational database developed under the Apache License, Version 2.0 [37] [38]. Karaf is a modular container as an open source runtime environment supporting the standard OSGi [39]. XNIO is a non-blocking I/O layer and library used to build efficient networking applications [40].

Test inputs. For measuring IPC of the chosen subjects, we had to execute them against varied test inputs. For all subjects, we constructed an integration test case according to the official user guide available on respective project website, as elaborated below. Creation of these integration tests did not need deep knowledge about each system because the descriptions on respective user guides are straightforward and clear. In addition, we utilized two types of test for ZooKeeper that came within the project package: load test and system test. For each integration test, we started two to five server and client nodes on different machines, except for ZooKeeper we started an additional node as a container. For peer-to-peer systems (e.g., OpenChord), we operated on all nodes; for others, we operated only on the client (which then automatically triggered operations on other nodes).

Specifically, for OpenChord, the operations were: we first created an overlay network on the first node; next, we joined the network on other two nodes, inserted a new data entry to the network on the third node, looked up and deleted the data entry on the first node; lastly, we listed all data entries on second node. For ZooKeeper, we first created two nodes, looked up for both, checked their attributes, changed the data association between them, and then deleted both nodes. For Voldemort, the client operations were adding a key-value pair, querying the value of the key, deleting the key, and retrieving the pair again. For Derby, we searched all the data records (`SELECT *`) from a relational database (one table) created beforehand. For Karaf, we created a container hosted by the server and then executed two operations in order: list all packages (`la`), listing OSGi bundles (`list`).

The four remaining subjects, Thrift, xSocket, Netty and XNIO, are all libraries/frameworks, for which we needed to instantiate them by developing applications. For Thrift, we used its libraries to develop a calculator consisting of a server and a client component. (The Thrift file must be transferred to Java programs first.) We performed against the calculator (from its client) basic arithmetics (addition, subtraction, multiplication, and division). For xSocket, we first started one server instance and two client instances. Next, we sent one text message from one client, and a different message from the other client, to the server. For both Netty and XNIO, we started one server and one client instance, and then sent one message from the client to the server.

B. *Quality Factors*

We hypothesized that six quality factors might be informed by IPC measurement: maintainability, correctness, (code) stability, understandability, security, and performance [1]. To enable our investigation of how these quality factors may be related to our IPC metrics, we needed to for each quality factor (1) identify a direct measure to quantify it, and (2) actually measure it directly by computing the quantity. Next, we elaborate each of these factors regarding both needs.

Maintainability. We characterize maintainability of a system via the average interval of a maintenance cycle of the system, referred to as *release interval*. The rationale is that release interval can directly measure maintainability—intuitively, a shorter (longer) interval means a higher (lower) frequency of changes, suggesting that the system needs more (less) maintenance efforts hence lower (higher) maturity of the software. The quantity is computed as the average time span (number of days) between two consecutive software versions (releases). To account for the effect of software size on maintainability, we further normalize it by computing the release interval per source line of code (SLOC)—dividing the average release interval by the average SLOC across all the historical versions.

To compute the (normalized) release interval for each of the nine subjects, we collected the entire release history accessible to us through respective online resources, including the release date and source code of each version. Specifically, the history of OpenChord was obtained from SourceForge [41], the history of Thrift [42] and Derby [43] both from Apache Projects, the history of xSocket [44], Netty [45], Karaf [46] and XNIO [47] all from respective Maven repository, and the history of Voldemort [48] and ZooKeeper [49] from their GitHub repository. As an example of quantifying maintainability via release interval, Table III shows the version history of OpenChord, the size of each version, and accordingly the normalized release interval (0.024).

TABLE III
VERSION HISTORY AND MAINTAINABILITY OF OPENCHORD

Version	Release date	Time span	#SLOC
1.0	1/31/2006	- (initial release)	7,792
1.0.1	1/5/2007	339	7,839
1.0.2	1/12/2007	7	8,213
1.0.3	7/18/2007	187	8,343
1.0.4	10/25/2007	99	8,364
1.0.5	11/4/2008	376	9,244
	Average	**201.6**	**8,299.2**
		Release interval=201.6/8,299.2=0.024	

Correctness. We quantify the functional correctness of a system via *defect density* of the system—the number of defects normalized by the system's size in terms of KSLOC. This is an intuitive/immediate, reverse measure of correctness, a key qualify factor in almost all quality models [1]: the higher the defect density, the lower the correctness of the system.

To compute the defect density for each of our subjects, we collected the data on defects throughout the same version history we referred to for computing the release intervals. For OpenChord [50] and xSocket [51], the defects were collected from SourceForge. For Thrift [52], ZooKeeper [53], Derby [54] and Karaf [55], we found the defects in the Jira database. The defects of Volemort [56] and Netty [57] were obtained from respective Github issue collection. We gathered defect data for XNIO [58] from the JBoss project's Maven repository. We accumulated the total number of defects both for the entire subject and per relevant classes, according to varied bug repositories of each version of the subject. The system-level defect density was then obtained from the defect total and the average SLOC of the versions in the history. For example, we found 40 defects in the version history of Voldemort (81 releases in 5 years), of which 1 was in class `voldemort.client.ClientThreadPool` and 3 in total was responsible by class `voldemort.client.protocol.admin`. The average KSLOC of the subject is 103.185, thus the defect density is 40/103.185=0.388 (0.388 bugs in every thousand SLOC).

Stability. We define stability as a quality factor concerning how stable the codebase of a system is, which is quantified as the average number of source lines of code changed between two consecutive versions of the system, referred to as *code churn* [59]. It is then normalized by the size (SLOC) of the latter version. The system-level stability is measured as the

TABLE IV
CODE CHANGE HISTORY AND CODE-STABILITY (QUANTIFIED THROUGH CODE CHURN) OF THRIFT

Version	0.2.0	0.3.0	0.4.0	0.5.0	0.6.0	0.6.1	0.7.0	0.8.0	0.9.0	0.9.1	0.9.2	0.9.3	0.10.0	0.11.0
SLOC	6417	6,830	7,903	8,913	10,093	10,093	10,230	11,163	11,661	12,635	13,341	14,094	14,217	14,510
Diff size	-	984	909	296	1,828	126	179	650	189	543	596	179	749	209
Per-pair churn		0.1441	0.1150	0.0332	0.1811	0.0125	0.0175	0.0582	0.0162	0.0430	0.0447	0.0127	0.0527	0.0144

TABLE V
DIRECT MEASURES OF UNDERSTANDABILITY FOR ALL SUBJECTS

Subject	OpenChord	Thrift	xSocket	Voldemort	ZooKeeper	Netty	Derby	Karaf	XNIO
Cyclomatic Complexity	4,688	5,701	10,739	74,042	35,229	67,252	118,622	6,306	2,972
Understability	0.5071	0.3929	0.6814	0.7248	0.5664	0.4128	0.1723	0.1034	0.5627

average code churn over all pairs of consecutive versions in the release history of the system considered. This direct measure of stability is similar and complementary to release interval in that both look at the changes between consecutive software versions, but the former focuses on the scale of the changes (i.e., in the perspective of software itself) while the latter on the time spent for making those changes (i.e., in the perspective of human/developer cost). Accordingly, the same dataset used for quantifying release interval was used for computing the code churn of each subject. To illustrate our direct measure of stability, Table IV shows the change history of Thrift, including the version number (first row), size (SLOC) of each version (second row), number of changed lines of code relative to the previous version (third row), and the code churn for each pair of versions. The system-level code churn is the average of the per-pair churns, which is 0.0573 in this example.

Understandability. We quantify understandability of a system using the McCabe's cyclomatic complexity metric [60], normalized by the size (SLOC) of the system. We used the same dataset as used for computing the dynamic IPC metrics—only one version of each subject (as listed in Table II) was measured. Table V gives the direct measure of *understandability* of all the nine subjects studied, including the raw cyclomatic complexity (second row) and the normalized complexity as the direct measure (last row).

Security. We directly measure the dynamic information flow security, a critical aspect of system security, in terms of two characteristics (metrics) of *taint flow paths* in the analyzed execution: *path count*, which is the total number of such paths, and *path length*, which is the average length of the paths. The length of a taint flow path is the number of statements on the path. As for quantifying understandability, we computed these two security metrics for a single version of each of the nine subjects (see Table II). We also normalized both metrics by the subject size in terms of KSLOC. Table VI gives our security measures of all subjects with respect to the executions generated using the chosen test inputs (Table II). For subjects (e.g., Derby) and test types (e.g., system test of ZooKeeper) not listed in the Table VI, we found no taint flow paths (thus their security measures were all zeros).

TABLE VI
DIRECT MEASURES OF SECURITY FOR ALL SUBJECTS

Subject	Thrift	xSocket	Voldemort	ZooKeeper-load	Netty
raw path count	3	2	42	64	2
normalized path count	**0.207**	**0.127**	**0.411**	**1.029**	**0.012**
raw path length	171.0	54.5	362.2	466.5	2,476.5
normalized path length	**11.785**	**3.458**	**3.545**	**7.501**	**15.200**

Performance. IPC characteristics might be connected to system performance. We characterize the performance of a system in terms of *operation speed*, quantified as the number of operations the system can perform per second, normalized by the subject size in terms of SLOC. Instead of computing these direct measures, we collected the performance data from relevant online sources of each subject. For xSocket, the relative performance data, compared to Netty's performance data [61], came from [62]. We then derived the performance data of xSocket in absolute terms from these sources together. For other subjects, we obtained the data immediately from [63] for OpenChord, from GitHub [64] for Thrift, from its project website [65] for Voldemort, from the Confluence site [66] for Zookeeper, from [67] for Derby, and from [68] for Karaf. We did not find performance data for XNIO, thus omitted this subject in our correlation analysis between IPC metrics and performance. The direct measures of system performance (via operation speed) for other subjects are listed in Table VIII.

C. Research Questions and Procedure

We list the research questions that guided our study and outline experimental procedures for answering the questions.

- **RQ1: How are the processes coupled in distributed systems with respect to the proposed IPC metrics?** We characterize the IPC coupling and relevant other traits of distributed systems in their typical operation profiles, by measuring IPC in those executions with respect to the proposed IPC metrics. We computed all the IPC metrics (§ III) against all the subjects and tests (Table II), using our dynamic dependence abstraction framework [19]. The goal of answering this question is to demonstrate the practicality of IPC measurement with our approach.
- **RQ2: Are the proposed IPC metrics related to any of the six quality factors?** With the IPC measures computed for RQ1 and the direct measures of these quality factors computed as detailed above (§ IV-B), we performed extensive correlation analysis to examine the correlation between each of the four IPC metrics and each of the six quality factors. We used Spearman's correlation coefficients [20] to quantify the correlation, a non-parametric method making no assumptions about the relationship between the two variables involved (which is the reason we chose Spearman's method over alternatives like Kendall's [69] and Pearson's [70] methods). Further, we adopted the interpretations of correlation strength according to varied value ranges of the Spearman coefficient r in [71]: the correlation is *very weak* if $|r|$ is below 0.20, *weak* if $|r|$ is between 0.20 and 0.39, *moderate* if $|r|$ is between 0.40 and 0.59, *strong* if $|r|$ is between 0.60 and 0.79, *very strong* if $|r|$ is 0.8 or above.

In addition, we used the `cloc` tool [72] to compute the size of a system (SLOC) and the `diff` tool [73] to compute the code differences between two versions. When quantifying a quality factor, we choose SLOC or KSLOC as the normalizing unit according to the value range of the quality factor's measure to avoid the metric value to be too small thus seem negligible. To compute the two security measures, we used our dynamic taint analysis tool for distributed programs [74] to produce taint flow paths, for which the lists of taint sources and sinks were curated based on the documentation of Java security/-cryptography APIs. We used a Java source code measurement tool JavaNCSS [75] to compute the cyclomatic complexity as a reversed metric of understandability (higher complexity implies lower understandability). All the machines used in our study were Ubuntu Linux 16.04.3 LTS workstations with an Intel E7-4860 2.27GHz CPU and 16GB DRAM.

V. RESULTS AND ANALYSIS

This section presents our major results and conclusions in response to the two major research questions.

A. RQ1: IPC Measurements

TABLE VII
IPC MEASUREMENT RESULTS (WITH SYSTEM-LEVEL IPC METRICS)

Subject+test type	RMC	RCC	CCC	IPR
OpenChord	3.89	41.12	0.72	0.1
Thrift	0.5	20.26	0.44	0.27
xSocket	0.33	35.27	0.61	0.14
Zookeeper-integration	13	84.5	0.49	0.09
Zookeeper-load	815.17	83	0.76	0.19
Zookeeper-system	71.67	26.67	0.72	0.32
Voldemort	25.50	119.00	0.79	0.11
Netty	0.67	148.04	0.70	0.43
Derby	1	49.7	1.25	0.28
Karaf	1	12	1.00	0.82
XNIO	1	44.98	0.86	0.43

Our results on IPC measurements are summarized in Table VII. Each number represents one of the four proposed IPC metrics computed for one subject and one type of test. For all subjects but ZooKeeper, the test type is omitted (first column) as only one integration test was considered for them.

The RMC results show that the two enterprise-scale systems, ZooKeeper and Voldemort, had medium to high degrees of message coupling among the distributed processes. In particular, with the executions against system and load tests, ZooKeeper saw an overwhelmingly higher RMC than did other subjects. A main reason is that for all other cases, the execution was driven by an integration test, which was relatively simple (mainly just between client and server processes). Especially, during the load test, ZooKeeper took an extraordinary level of workload leading to a large number of message exchanges among the processes; further, the load test needed all the processes to collaborate closely, hence a high RMC as a result. Intuitively, the RMC measures immediately indicate the complexity of IPC in terms of message passing among processes. The higher RMC metric values also implied heavier network communication costs. We also observed that the RMC values did not seem to be related to the system size (in terms of SLOC)—the processes in the largest system (Derby) were very lightly coupled, less than those in one of the smallest system (xSocket), despite both being with respect to the same type of test (i.e., integration test).

TABLE VIII
DIRECT MEASURES OF OTHER QUALITY FACTORS

Subject	Maintainability	Correctness	Stability	Performance
OpenChord	2.43E-02	6.14E-02	6.14E-02	1.32E-01
Thrift	2.54E-02	4.00E-03	6.15E-02	2.12E+00
xSocket	2.23E-03	3.81E-04	3.63E-02	8.70E-01
Zookeeper	1.85E-04	3.23E-04	2.41E-03	1.56E-01
Voldemort	3.56E-03	6.03E-03	1.01E-02	1.71E-01
Netty	4.38E-04	3.27E-03	9.19E-03	1.47E-01
Derby	4.18E-04	5.96E-03	2.28E-02	2.00E-02
Karaf	1.29E-03	3.56E-03	1.99E-02	1.75E-01
XNIO	3.24E-02	4.27E-03	6.93E-02	-

The RCC measures appeared to be independent of RMC, evidenced by the absence of association between the two metrics: the RCC of ZooKeeper-load, which had the highest RMC, had a relatively low yet not the lowest RCC; overall, higher/lower RCC was not associated with higher/lower RMC. Similarly to RMC, RCC saw no correlation with system size either. The numbers revealed that during the integration test, Netty had the most highly coupled processes at class level, followed by Voldemort, while in Karaf a class in one process generally did not influence much the classes in other processes.

Like RMC and RCC, CCC and IPR saw no correlation with subject size nor with any other IPC metrics. An implication of the lack of association among the four metrics is that each of them is uniquely informative/significant—none of them subsumed any others. What was obviously different between RMC/RCC and CCC/IPR was that the 11 measures saw substantial variations in RMC/RCC, yet very small variations in CCC/IPR were observed. This contrast suggests that RMC/RCC can vary widely from system to system, while CCC/IPR may be mostly in a narrow range. In terms of the numbers, the mean CCC was mostly around 1, meaning that every class collaborated with about one other class in a remote process. Complementary to the other three metrics that concern the dependence/reliance of one process on others (albeit at varied levels of granularity), IPR informs the *common* dependence of processes, That said, Karaf's processes had much higher degree of common functionalities shared among them than other systems; in contrast, the two processes in ZooKeeper's integration test had very little sharing.

Conclusion. The four IPC metrics each measured/characterized the IPC coupling in a uniquely informative/significant perspective—no one metric subsumes/implies another. Meanwhile, RCC and RMC can be sharply different among different systems while CCC and IPR seem to be relatively stable across systems. Even with the same type of execution scenarios, the coupling among distributed processes had no correlation with system sizes. The degree of coupling seems to generally follow a normal distribution: few systems saw extremely high or low coupling, and most systems had moderately coupled processes.

B. RQ2: Relation to Quality Factors

The data groups underlying our correlation analysis include (1) the dynamic measures with respect to each of the 4 IPC metrics over all subjects (all shown in Table VII) and (2) the direct measures of each of the six quality factors (in 7 metrics—security has two metrics and other factors each has one metric) over all subjects. Part of (2), on understandability and security, is shown in Table V and Table VI, respectively.

TABLE IX
CORRELATIONS BETWEEN IPC METRICS AND QUALITY FACTORS

		IPC metrics			
		RMC	RCC	CCC (mean)	IPR
Quality factors	Maintainability	-0.237	-0.383	-0.133	-0.151
	Correctness	0.017	0.167	**0.483**	0.276
	Stability	**-0.424**	**-0.583**	-0.017	-0.025
	Understandability	0.305	0.383	**0.467**	**-0.586**
	Security (path count)	0.159	0.348	**0.618**	**-0.450**
	Security (path length)	0.195	**0.453**	**0.583**	-0.371
	Performance	-0.390	**-0.467**	**-0.619**	-0.142

The remaining parts of (2) are summarized in Table VIII. Since we examine the correlation between each IPC metric and each quality factor, we performed the correlation analysis for 28 pairs of (1) and (2) data groups.

Note that the two groups in each pair need to be of equal sizes. Regarding the security factor, each group had 11 data points since we had the *dynamic* measures of security for each of the 11 cases (corresponding to the Rows 2–12 of Table VII). For other quality factors, since the measures were *static* (one measure per subject), the group on each of these factors had 9 data points, we had to use one aggregate IPC measure for each subject as well. To that end, we took the mean of the measures per IPC metric over the three executions of ZooKeeper (corresponding to the three test types). There was still an exception with the performance factor, though: we omitted XNIO as we did not have the performance data available for this subject, thus we simply took this subject out from both groups hence each group had 8 data points. Table IX lists the 28 resulting Spearman coefficients.

Our results show widely varying relations of our IPC metrics to the six quality factors. We regard a moderate or stronger correlation as *significant*, and marked such cases in boldface. All the four IPC metrics were negatively correlated with **maintainability** (quantified via release interval), which indicates higher IPC coupling being associated with lower maintainability. Recall that the IPC coupling is an indicator of implicit coupling among distributed components (which are decoupled by design). Among more coupled components/processes, changing one more likely affects others, contributing to more frequent maintenance (hence shorter release interval). Yet the correlation was mostly very weak and only RCC was close to a significant correlation with maintainability.

CCC was the only IPC metric that was significantly correlated to **correctness**, quantified by a reverse measure (defect density)—the higher the measure the lower the correctness. The correlation was positive because when a system has more classes in one component/process more coupled with classes in other components/processes, defects in one potentially propagate more broadly, hence a higher defect density.

Regarding **stability** (in terms of code churn size), the significant correlation with RMC and RCC implies that interprocess coupling may well indicate the size of code changes in each maintenance iteration. The negativity of the correlations further suggests that having components more coupled may actually reduce the amount of code changes. A plausible reason is that more changes in one component/process may more readily propagate to other components/processes automatically, hence less changes needed for the entire system in one maintenance cycle. The other two IPC metrics were negligible in terms of their relation to stability.

We observed that higher coupling in terms of one process/component's dependencies on others (i.e., RMC, RCC, and CCC) was consistently associated with higher cyclomatic complexity hence lower **understandability**, as evidenced by the positive correlations between this quality factor and the three metrics (especially CCC, with which the correlation was significant). The reason is that those dependencies intuitively made the system more complicated, hence more difficult, to understand. Interestingly, having more common functionalities among components/processes reduced the complexity hence enhanced understandability, according to the significant, negative correlation with IPR. The IPC metrics were similarly informative of **security**, with respect to both direct security measures: the more interprocess dependencies (measured by RMC/RCC/CCC) were related to more and longer taint flow paths (lower security), yet more sharing of functionalities (measured by IPR) related to fewer and shorted vulnerable paths (higher security). In particular, CCC was strongly informative about security as measured by the number of vulnerable paths.

The negative correlations of all the IPC metrics with **performance** indicate that a higher degree of interactivity among processes/components, in terms of either dependencies on one another (RMC/RCC/CCC) or dependencies in common (IPR), was consistently related to a lower operation speed (hence a lower level of performance). Justifiably, given that a system needs collaborations among its processes/components to finish a task, when one class/component has to communicate with more others across processes (as implied by the higher interactivity), it takes longer to carry out the task. ICCC was again the most informative metric for performance, as per its strongest correlation with this quality factor.

Conclusion. All the four IPC metrics were related to and informative of all the six quality factors, despite the variations in correlation strengths. In particular, higher IPC coupling in terms of dependencies of one process/component on others (via higher RMC/RCC/CCC) was significantly associated with higher defect density (lower correctness), smaller code churn (higher stability), higher complexity (lower understandability), more/longer vulnerable execution paths (lower security), and lower operation speed (lower performance). More sharing of functionalities among processes (via higher IPR) was significantly associated with higher understandability and security.

C. Threats to Validity

The main threat to *internal* validity lies in possible implementation errors in our computation for IPC metrics, direct measures of quality factors, and correlation analysis. To reduce this threat, we did a careful review for all our tools and used the two smallest subjects to manually validate their functionalities and analysis results. In addition, the validity of our results rely on that of the data we collected from various sources used for quantifying the six quality factors.

The main threat to *external* validity is that our study results may not generalize to other distributed systems and executions. To reduce this threat, we have chosen subjects of various sizes and application domains, focusing on real-world/industry-scale systems of varied architectures. We also considered different types of inputs, including integration, system, and load tests. Nevertheless, our results are best interpreted with respect to the systems and executions actually studied. Also, the number of subjects we considered immediately limited the number of data points included in each group in our correlation analyses hence the validity of the analysis results.

The main threat to *construct* validity lies in our use of statistical analysis for drawing our conclusions. In computing the system-level IPC metrics, we took the means of lower-level metrics without accounting for the variations (e.g., standard deviations), which would need to be considered in more thorough measurement schemes. To reduce the threat as regards our correlation analysis, we carefully chose Spearman's method over alternatives as it is a non-parametric method without assuming normality of underlying data's distribution or relationships between the data groups. Meanwhile, our correlation analysis looked at system-level measures; as a result, it involved a relatively small number of data points, which constitutes another threat to the validity of our results.

D. Lessons Learned And Recommendations

Our exploration of IPC metrics not only demonstrated the practicality of measuring IPC in large, real-world distributed systems, but also revealed the substantial presence (albeit with varying degrees) of *implicit* coupling among distributed components (that are generally decoupled in architectural design). And we showed that one way to reveal such implicit coupling is through measuring interprocess coupling.

Our results on IPC measurements revealed that higher coupling in terms of inter-process/component dependencies is generally bad for quality (by being significantly indicative of lower quality with respect to five out of the six factors considered). This largely confirmed the drawbacks of high coupling in general [2]. On the other hand, functional overlapping among distributed components can benefit quality by enhancing understandability and lowering security vulnerabilities. Note that CCC, RCC, IPR, and RMC was significantly correlated to four, three, two, and one out of the six quality factors, respectively, while this is also nearly a non-ascending order of granularity for the IPC measurement. Thus, our study suggests that a finer-grained metric may be more indicative of varied quality factors hence more useful for analyzing distributed system quality.

Based on our investigations, distributed system developers are recommended to attain and maintain a low degree of (implicit) coupling among systems components in order to achieve and sustain high quality. Developers should also promote code reuse among distributed components in light of its benefits for understandability and security of distributed software. Meanwhile, developers can leverage IPC measurements to understand/analyze hence improve various quality factors. IPC metrics can also be used to predict direct quality measures according to their correlation (we have validated this on Thrift for predicting maintainability and understandability using RCC metric, and omitted the details for space limit).

VI. Related Work

Dynamic coupling metrics have been well studied for centralized systems [2], [4], [5]. Arisholm et al. [13] defined a set of dynamic coupling metrics for object-oriented software and studied the relationship between dynamic coupling measures and software change-proneness. Their dynamic class export/import coupling (IC_CD and EC_CD) metrics initially motivated the definition of our RMC metric. Compared to their coupling measurement between *classes*, the RMC metric measures the coupling between *processes*.

Dynamic coupling metrics also have been used to estimate architectural risks [7] and complexity [14] in relation to quality factors such as maintainability [8], [76]. Most of these metrics were defined under the assumption that there exists an explicit reference/invocation between the entities (e.g., object, method, class, etc.) involved in the coupling measure, thus they are not suitable for measuring interprocess communication in distributed systems. On the other hand, however, our IPC metrics can also be used as complexity metrics and as indicators of a variety of quality factors.

In [17], the authors defined a dynamic component coupling metric (CP) directly based on inter-component dependencies derived from method executions with timing information. Conceptually, the CP metric is closely related to our IPR metric, in that both are based on approximated dynamic dependencies across components. However, the interprocess dependencies on which our IPR computation is based are significantly more precise than the purely control-flow-based dependencies approximated in [17], according to our previous study [24]. In addition, CP was defined for measuring structural complexity, while IPR is proposed primarily as a reusability metric. Previous reuse metrics mainly concern reusing library code and connectivity between server and client nodes as a whole [77]. Instead, we measure interprocess reusability at code level.

VII. Conclusion

This paper contributed to dynamic software measurement with a novel set of four metrics for distributed systems that characterize their IPC, a vital aspect of distributed systems' run-time behaviors, at various levels (whole-system, process/-component, class, and method). In addition, with these metrics, we measured the IPC in nine real-world distributed systems with respect to varied execution scenarios, and demonstrated that each of the metrics was uniquely informative of IPC traits in those systems and their executions. We further demonstrated the implications of the IPC measurements to distributed software quality and their practical usefulness for understanding six quality factors, which were quantified through direct measures computed from relevant datasets we collected from diverse sources. Our study revealed that IPC measures can be significantly indicative of the various quality aspects of distributed systems hence potentially help developers understand, assess, and hence improve the quality of these systems. Future work will focus on exploring how IPC *coupling* measures can be leveraged for predicting different quality factors that are difficult to measure directly, while expanding the scope to include *cohesion* metrics of distributed systems.

VIII. Acknowledgment

We thank the anonymous reviewers for their thoughtful comments, which greatly helped us improve this paper.

REFERENCES

[1] D. Galin, *Software quality assurance: from theory to implementation*. Pearson Education India, 2004.

[2] S. H. Kan, *Metrics and models in software quality engineering*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[3] N. Fenton and J. Bieman, *Software Metrics: A Rigorous and Practical Approach*. CRC Press, 2014.

[4] A. Tahir and S. G. MacDonell, "A systematic mapping study on dynamic metrics and software quality," in *Proceedings of IEEE International Conference on Software Maintenance*. IEEE, 2012, pp. 326–335.

[5] J. K. Chhabra and V. Gupta, "A survey of dynamic software metrics," *Journal of Computer Science and Technology*, vol. 25, no. 5, pp. 1016–1029, 2010.

[6] B. Dufour, K. Driesen, L. Hendren, and C. Verbrugge, "Dynamic metrics for Java," in *ACM SIGPLAN Notices*, vol. 38, no. 11. ACM, 2003, pp. 149–168.

[7] S. M. Yacoub and H. H. Ammar, "A methodology for architecture-level reliability risk analysis," *IEEE Transactions on Software Engineering*, vol. 28, no. 6, pp. 529–547, 2002.

[8] A. Gosain and G. Sharma, "Predicting software maintainability using object oriented dynamic complexity measures," in *International Conference on Smart Trends for Information Technology and Computer Communications*. Springer, 2016, pp. 218–230.

[9] T. Richner and S. Ducasse, "Recovering high-level views of object-oriented applications from static and dynamic information," in *Proceedings of IEEE International Conference on Software Maintenance*. IEEE, 1999, pp. 13–22.

[10] R. Geetika and P. Singh, "Empirical investigation into static and dynamic coupling metrics," *ACM SIGSOFT Software Engineering Notes*, vol. 39, no. 1, pp. 1–8, 2014.

[11] I. Beschastnikh, P. Wang, Y. Brun, and M. D. Ernst, "Debugging distributed systems," *Queue*, vol. 14, no. 2, p. 50, 2016.

[12] G. Coulouris, J. Dollimore, T. Kindberg, and G. Blair, *Distributed Systems: Concepts and Design*, 5th ed. Addison-Wesley Publishing Company, 2011.

[13] E. Arisholm, L. C. Briand, and A. Foyen, "Dynamic coupling measurement for object-oriented software," *IEEE Transactions on Software Engineering*, vol. 30, no. 8, pp. 491–506, 2004.

[14] A. Gosain and G. Sharma, "Object-oriented dynamic complexity measures for software understandability," *Innovations in Systems and Software Engineering*, vol. 13, no. 2-3, pp. 177–190, 2017.

[15] D. Wybranietz and D. Haban, "Monitoring and performance measuring distributed systems during operation," *ACM SIGMETRICS Performance Evaluation Review*, vol. 16, no. 1, pp. 197–206, 1988.

[16] L. Wang and J. Kangasharju, "Measuring large-scale distributed systems: case of bittorrent mainline dht," in *Peer-to-Peer Computing (P2P), 2013 IEEE Thirteenth International Conference on*. IEEE, 2013, pp. 1–10.

[17] W. Jin, T. Liu, Y. Qu, Q. Zheng, D. Cui, and J. Chi, "Dynamic structure measurement for distributed software," *Software Quality Journal*, pp. 1–27, 2017.

[18] H. Cai and D. Thain, "Distea: Efficient dynamic impact analysis for distributed systems," *arXiv preprint arXiv:1604.04638*, 2016.

[19] H. Cai and X. Fu, "D^{2}ABS: A framework for dynamic dependence abstraction of distributed programs," Washington State University, technical report EECS-2019-01-19, January 2019.

[20] Wikipedia, "Spearman's rank correlation coefficient," https://en.wikipedia.org/wiki/Spearman%27s_rank_correlation_coefficient, 2019.

[21] X. Fu and H. Cai, "Artefact package on IPC measurement of distributed systems," https://www.dropbox.com/s/oddfgntbkjrf69l/DistMeasure_artefact.zip?dl=0, 2019.

[22] M. Sharifi, E. M. Khaneghah, M. Kashyian, and S. L. Mirtaheri, "A platform independent distributed ipc mechanism in support of programming heterogeneous distributed systems," *The Journal of Supercomputing*, vol. 59, no. 1, pp. 548–567, 2012.

[23] J. Jenkins and H. Cai, "ICC-Inspect: supporting runtime inspection of Android inter-component communications," in *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems*. ACM, 2018, pp. 80–83.

[24] H. Cai and D. Thain, "DistIA: A cost-effective dynamic impact analysis for distributed programs," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, 2016, pp. 344–355.

[25] H. Cai and R. Santelices, "Diver: Precise dynamic impact analysis using dependence-based trace pruning," in *Proceedings of International Conference on Automated Software Engineering*, 2014, pp. 343–348.

[26] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[27] J. Jenkins and H. Cai, "Dissecting Android inter-component communications via interactive visual explorations," in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2017, pp. 519–523.

[28] Apache, "ZooKeeper," https://zookeeper.apache.org/, 2015.

[29] H. Cai, "Hybrid program dependence approximation for effective dynamic impact prediction," *IEEE Transactions on Software Engineering*, vol. 44, no. 4, pp. 334–364, 2018.

[30] P. V. Marsden, "Recent developments in network measurement," *Models and Methods in Social Network Analysis*, vol. 8, p. 30, 2005.

[31] Bamberg University, "Open Chord," http://sourceforge.net/projects/open-chord/, 2015.

[32] Apache, "Thrift," https://thrift.apache.org/, 2018.

[33] Vice, "xSocket," http://xsocket.org/, 2018.

[34] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "Zookeeper: Wait-free coordination for internet-scale systems." in *Proceedings of USENIX Annual Technical Conference*, vol. 8, 2010, p. 9.

[35] Apache, "Voldemort," https://github.com/voldemort, 2015.

[36] Wikipedia, "Netty(software)," https://en.wikipedia.org/wiki/Netty_(software), 2018.

[37] ——, "Apache Derby," https://en.wikipedia.org/wiki/Apache_Derby, 2018.

[38] Apache, “Apache Derby,” https://db.apache.org/derby/, 2018.

[39] Karaf, “Apache Karaf,” https://karaf.apache.org/, 2019.

[40] XNIO - JBoss Community, “XNIO,” http://xnio.jboss.org/, 2012.

[41] SourceForge, “Open Chord,” https://sourceforge.net/projects/open-chord/files/Open%20Chord%201.0/, 2008.

[42] Apache, “Index of /dist/thrift,” http://archive.apache.org/dist/thrift/, 2018.

[43] ——, “Apache Derby,” https://db.apache.org/derby/derby_downloads.html, 2019.

[44] Maven Repository, “XSocket,” https://mvnrepository.com/artifact/org.xsocket/xSocket, 2008.

[45] ——, “Netty All In One,” https://mvnrepository.com/artifact/io.netty/netty-all, 2019.

[46] ——, “Apache Karaf,” https://mvnrepository.com/artifact/org.apache.karaf/karaf, 2018.

[47] ——, “XNIO API,” https://mvnrepository.com/artifact/org.jboss.xnio/xnio-api, 2018.

[48] Github, “Voldemort,” http://https://github.com/voldemort/voldemort/releases, 2017.

[49] Apache, “Apache Zookeeper,” https://github.com/apache/zookeeper/releases, 2019.

[50] SourceForge, “Open Chord bugs,” https://sourceforge.net/p/open-chord/bugs/, 2008.

[51] Maven Repository, “XSocket bugs,” https://mvnrepository.com/artifact/org.xsocket/xSocket, 2008.

[52] Apache, “Thrift bugs,” http://archive.apache.org/dist/thrift/, 2018.

[53] ——, “Zookeeper bugs,” https://github.com/apache/zookeeper/releases, 2019.

[54] ——, “Derby bugs,” https://db.apache.org/derby/derby_downloads.html, 2019.

[55] Maven Repository, “Apache Karaf bugs,” https://mvnrepository.com/artifact/org.apache.karaf/karaf, 2018.

[56] Github, “Voldemort bugs,” http://https://github.com/voldemort/voldemort/releases, 2017.

[57] Maven Repository, “Netty bugs,” https://mvnrepository.com/artifact/io.netty/netty-all, 2019.

[58] ——, “XNIO API bugs,” https://mvnrepository.com/artifact/org.jboss.xnio/xnio-api, 2018.

[59] G. A. Hall and J. C. Munson, “Software evolution: code delta and code churn,” *Journal of Systems and Software*, vol. 54, no. 2, pp. 111–118, 2000.

[60] T. J. McCabe, “A complexity measure,” *IEEE Transactions on Software Engineering*, no. 4, pp. 308–320, 1976.

[61] Y. Wang, L. Huang, X. Liu, T. Sun, and K. Lei, “Performance comparison and evaluation of websocket frameworks: Netty, undertow, vert. x, grizzly and jetty,” in *2018 1st IEEE International Conference on Hot Information-Centric Networking (HotICN)*. IEEE, 2018, pp. 13–17.

[62] M. Hammerton, J. Trevathan, T. Myers, and W. Read, “Optimising data transmission in heterogeneous sensor networks,” *International Journal of Information*, vol. 7, no. 9, 2013.

[63] K. Waagan, “Building a replicated data store using berkeley db and the chord dht,” Master’s thesis, Institutt for datateknikk og informasjonsvitenskap, 2005.

[64] Github, “smallnest/RPC-TEST,” https://github.com/smallnest/RPC-TEST, 2015.

[65] “Project Voldemort,” https://www.project-voldemort.com/voldemort/performance.html, 2017.

[66] Confluence, “ServiceLatencyOverview,” https://cwiki.apache.org/confluence/display/ZOOKEEPER/ServiceLatencyOverview, 2019.

[67] D. Oracle Berkeley, “Java edition vs. apache derby: A performance comparison,” 2006.

[68] E. Medvedeva, “Performance comparison of jboss integration platform implementations,” Ph.D. dissertation, Masarykova univerzita, Fakulta informatiky, 2014.

[69] Wikipedia, “Kendall rank correlation coefficient,” https://en.wikipedia.org/wiki/Kendall_rank_correlation_coefficient, 2018.

[70] ——, “Pearson correlation coefficient,” https://en.wikipedia.org/wiki/Pearson_correlation_coefficient, 2019.

[71] I. Weir, “Spearmans correlation,” vol. 29, 2016, http://www.statstutor.ac.uk/resources/uploaded/spearmans.pdf.

[72] cloc.org, “Counting source lines of code,” cloc.sourceforge.net/, 2019.

[73] GNU, “compare files line by line,” man7.org/linux/man-pages/man1/diff.1.html/, 2019.

[74] X. Fu and H. Cai, “DistTaint: Application-level dynamic information flow analysis for distributed systems,” Washington State University, technical report EECS-2019-02-20, February 2019.

[75] “JavaNCSS - A Source Measurement Suite for Java,” https://www2.informatik.hu-berlin.de/swt/intkoop/jcse/tools/JavaNCSS%20-%20A%20Source%20Measurement%20Suite%20for%20Java.html, 2012.

[76] M. Perepletchikov and C. Ryan, “A controlled experiment for evaluating the impact of coupling on the maintainability of service-oriented software,” *IEEE Transactions on Software Engineering*, vol. 37, no. 4, pp. 449–465, 2011.

[77] W. Frakes and C. Terry, “Software reuse: metrics and models,” *ACM Computing Surveys (CSUR)*, vol. 28, no. 2, pp. 415–435, 1996.