

Server for SQLite Database: Multithreaded HTTP Server with Synchronized Database Access and JSON Data-Interchange

Noprianto*, Benfano Soewito**, Sani Muhamad Isa**, Karto Iskandar*, Ford Lumban Gaol*, Raymond Kosala***

*Doctor of Computer Science, Bina Nusantara University, Jakarta, Indonesia

**Master in Computer Science, Bina Nusantara University, Jakarta, Indonesia

***School of Computer Science, Bina Nusantara University, Jakarta, Indonesia

`nop@noprianto.com`, `bsoewito@binus.edu`, `sisab@binus.edu`, `karto_i@binus.edu`, `fgaol@binus.edu`,
`rkosala@binus.edu`

Abstract— *SQLite database engine is small, fast, reliable, and does not need a server to work. And, if needed, multiple applications may work with same SQLite database, optionally, using filesystem sharing services. However, file sharing option is not always available and reliable in certain conditions. In this paper, we propose a server for SQLite database. This server will be using HTTP as communication protocol. To enable multiuser application, this server will be multithreaded. However, access to the database file will be synchronized to help prevent database corruption. And, as data-interchange format, JSON will be used. We also run several tests to make sure that proposed server is usable. Test results show that the database file is not corrupted, while still maintain acceptable performance.*

Keywords— *Embedded database, database server, HTTP server, concurrent programming, multithreaded application*

I. INTRODUCTION

SQLite is a serverless database engine that works with single database file [1]. Being serverless, there is no database server to configure, and that makes software development and deployment easier. In order to work with a database, application only need to access the database file directly. And thanks to single file feature, there is only one file to access and backup.

SQLite is also lightweight and small [1]. With database engine less than 500 KiB, and embeddable into application, this database engine is virtually runnable everywhere, from less-powered embedded devices to enterprise servers.

To make it more useful, SQLite is also thread-safe and reliable [2]. Being serverless and single file do not prevent SQLite database to be accessed from multiple operating system processes and/or threads. Combined with file sharing service, this makes simple multi-user application possible (even if the application is not designed to do so).

And, as the name might tell us: this database engine is SQL standard compatible and offers support for a large subset of SQL92 [3]. That way, data can be manipulated using industry-standard language. There is no custom, in-house, and/or non-standard way to work with SQLite. All those SQL knowledge is applicable here, too (with few adjustments when needed).

Plus, there is no license fee to use, modify, or even sell modified versions of SQLite. This is because SQLite is public domain software, and no claim of ownership is made to any part of the code [2].

Combined with its amazing technical features, this makes inclusion of this database engine – in free or proprietary products – a very interesting option. Long story short, SQLite – started in 2000-05-09 – is the most widely deployed database in the world and used more than all other database engines combined [4].

As satisfied users, we are happy to see its continuous development and how the development reflects the intent of the developers to support SQLite through the year 2050 [1]. And, to keep it lite, we understand that some non-core features must be kept away. And, when we need these features, for whatever reasons, we prefer to develop a separate application rather than modify the engine, as this will make it harder to sync with future developments.

One of features that we need is a server for SQLite. This might sound weird because SQLite is a serverless database. And, there are many free, cross-platform, proven relational database engines that work in client/server architecture. From free/open source to expensive proprietary products. From simple and small engines to the most complicated ones.

But we have our own reason. We need to integrate some applications that already work – and happy – with SQLite databases. And, since there are constraints in the resources, we prefer not to migrate to client/server database engine. Plus, what we do is only the integration part. For the worst scenario, we need a server that reads/writes the database file, which is already being accessed directly by another applications. And, for the best scenario, only that server has access to the database file (and another applications will be adapted to connect to this server, without accessing the database file directly).

This is because we do not want the server to corrupt the database. While we understand that SQLite is reliable [2], we want to make sure that, in that server, only one thread can access that database file, at a given time. Another clients are

still allowed to connect to that server, but they have to wait because access to the database file is synchronized.

Then, we have at least two questions. The first would be the communication protocol. And the last would be the data-interchange format. And we do not want to reinvent the wheel. For that purpose, we found that HTTP protocol is virtually everywhere and supported by many programming language runtime environments. HTTP also has been in use by the World-Wide Web since 1990 [5]. And, for data-interchange, we prefer JSON because it is faster and use fewer resources compared to XML [6]. There many libraries to work with JSON, available for many programming languages [7].

II. HTTP AND JSON

The Hypertext Transfer Protocol (HTTP) is a stateless application-level protocol for distributed, collaborative, hypertext information systems [8]. HTTP is a request and response protocol, where HTTP client sends a request to a HTTP server, and through some optional processing, that HTTP server send back the response. HTTP server is the server behind a website. Since its inception, there are some versions of HTTP. All of them are published as IETF Request For Comments: HTTP/1.0 RFC1945, HTTP/1.1 RFC2616, and HTTP/2 RFC7540.

JSON (JavaScript Object Notation) is a lightweight data-interchange format. It is based on a subset of the JavaScript Programming Language, Standard ECMA-262 3rd Edition - December 1999 [7]. JSON is published as IETF RFC4627 and ECMA-404 standard.

III. METHODOLOGY

In this paper, we want to test the feasibility of this multithreaded HTTP with synchronize database access approach. And our goal is to make sure that the database file is not corrupted after some testing scenarios. First of all, multithreaded HTTP server was developed, and it must work with single SQLite database file. If we need to work with multiple database files, then we have to start that HTTP server multiple times, on its own port number (default port number is 8080).

In this initial version, our HTTP server will handle only the GET request method. And only root (/) path is being handled at this time (more on this, later). In each request, server will expect at least one variable, named q, that should contain SQL command, encoded in base64. Another optional variable is c, that should present and non-empty if one expect that given SQL command will trigger any changes to be committed. Screen shot of server output is shown in figure 1.

Our focus is not in the multithreaded model itself. However, we try to propose a multithreaded and safe HTTP-based SQLite database server using JSON as data-interchange format.

```
QwOV8zNjgzKGEpIHZhbHVLcygyOSk=6c=1 HTTP/1.1" 200 -
127.0.0.1 - - [29/Oct/2016 16:46:53] "GET /?q=aw5zZXJ0IGludG8gdGFibGVfMTQ3NzczND
QwOV8zNjgzKGEpIHZhbHVLcygyMCK=6c=1 HTTP/1.1" 200 -
127.0.0.1 - - [29/Oct/2016 16:46:53] "GET /?q=aw5zZXJ0IGludG8gdGFibGVfMTQ3NzczND
QwOV8zNjgzKGEpIHZhbHVLcygyMSk=6c=1 HTTP/1.1" 200 -
127.0.0.1 - - [29/Oct/2016 16:46:53] "GET /?q=aw5zZXJ0IGludG8gdGFibGVfMTQ3NzczND
QwOV8zNjgzKGEpIHZhbHVLcygyMik=6c=1 HTTP/1.1" 200 -
127.0.0.1 - - [29/Oct/2016 16:46:53] "GET /?q=aw5zZXJ0IGludG8gdGFibGVfMTQ3NzczND
QwOV8zNjgzKGEpIHZhbHVLcygyNCK=6c=1 HTTP/1.1" 200 -
127.0.0.1 - - [29/Oct/2016 16:46:53] "GET /?q=aw5zZXJ0IGludG8gdGFibGVfMTQ3NzczND
QwOV8zNjgzKGEpIHZhbHVLcygyNSk=6c=1 HTTP/1.1" 200 -
127.0.0.1 - - [29/Oct/2016 16:46:53] "GET /?q=aw5zZXJ0IGludG8gdGFibGVfMTQ3NzczND
QwOV8zNjgzKGEpIHZhbHVLcygyNik=6c=1 HTTP/1.1" 200 -
127.0.0.1 - - [29/Oct/2016 16:46:53] "GET /?q=aw5zZXJ0IGludG8gdGFibGVfMTQ3NzczND
QwOV8zNjgzKGEpIHZhbHVLcygyNyK=6c=1 HTTP/1.1" 200 -
127.0.0.1 - - [29/Oct/2016 16:46:53] "GET /?q=aw5zZXJ0IGludG8gdGFibGVfMTQ3NzczND
QwOV8zNjgzKGEpIHZhbHVLcygyOck=6c=1 HTTP/1.1" 200 -
127.0.0.1 - - [29/Oct/2016 16:46:53] "GET /?q=aw5zZXJ0IGludG8gdGFibGVfMTQ3NzczND
QwOV8zNjgzKGEpIHZhbHVLcygyOSk=6c=1 HTTP/1.1" 200 -
127.0.0.1 - - [29/Oct/2016 16:46:53] "GET /?q=c2VsZWNOIGlhcHSt1dJRCkgZnJvSB0YW
JsZV8xNDc3NzMONDA5XzM2ODM= HTTP/1.1" 200 -
```

Figure 1. HTTP server is serving GET request to /, with q and c variables

As this server will handle concurrent connections, lock to the database must be acquired first to make sure that only one thread has exclusive access. After the database operation is done, server will release the lock, save the result as JSON format, and send it to the requesting client.

In this approach, if one SQL command take long time to complete, then others must wait. If we have one blocking SQL command, then the whole server will also block. As a simple workaround, we only allow one SQL command at a time. However, this will be discussed later.

This server is developed using Python programming language version 3.x (tested in version 3.4), using only standard library (no dependency to external libraries), in single Python script. There is no platform-specific codes, so it should be run unmodified in another operating systems, where modern Python version 3 runtime is installed. This is free/open source software, and its source code is available from <http://noprianto.com>.

IV. TESTING PROCEDURE

How the testing will be done? Since the server will work with concurrent connections, we will spawn a number of threads, and each of them will perform a number of database operation (both writing and reading). And this is applied to both empty and non-empty database. After the testing is done, we will check if the database is still in healthy state. Screen shot of testing output is shown in figure 2.

```
[Thread-16][Table: table_1477734409_8030][29]
[Thread-59][Table: table_1477734409_3683][16]
[Thread-59][Table: table_1477734409_3683][17]
[Thread-59][Table: table_1477734409_3683][18]
[Thread-59][Table: table_1477734409_3683][19]
[Thread-59][Table: table_1477734409_3683][20]
[Thread-59][Table: table_1477734409_3683][21]
[Thread-59][Table: table_1477734409_3683][22]
[Thread-59][Table: table_1477734409_3683][23]
[Thread-59][Table: table_1477734409_3683][24]
[Thread-59][Table: table_1477734409_3683][25]
[Thread-59][Table: table_1477734409_3683][26]
[Thread-59][Table: table_1477734409_3683][27]
[Thread-59][Table: table_1477734409_3683][28]
[Thread-59][Table: table_1477734409_3683][29]
[Thread-1][Table: table_1477734409_7064][response: {'data': [[30]]}]
[Thread-59][Table: table_1477734409_4756][response: {'data': [[30]]}]
[Thread-38][Table: table_1477734409_3077][response: {'data': [[29]]}]
[Thread-25][Table: table_1477734409_9909][response: {'data': [[30]]}]
[Thread-35][Table: table_1477734409_4456][response: {'data': [[30]]}]
[Thread-29][Table: table_1477734409_1602][response: {'data': [[30]]}]
[Thread-9][Table: table_1477734409_1656][response: {'data': [[30]]}]
[Thread-58][Table: table_1477734409_9980][response: {'data': [[30]]}]
```

Figure 2. Testing is being run, showing response from HTTP server

Our testings are done in one computer system, powered by 2.0 GHz CPU, with 8 GB of RAM. For the storage device, SSD is used, formatted with ext4 file system. Operating server used is GNU/Linux running kernel version 4.6. All software are developed with Python programming language version 3.4, using only standard library (no dependency to external libraries). All source codes for testing are also available in the same directory at code repository.

In our experiments, we spawned 60 threads, and each of them then create a table with random name, populate that table with 30 rows of integer value, and select maximum ROWID (unique row identifier) value from that table. We run this test several times. For non-empty database, we run this test on a database that contains 60, 120, and 180 tables. SQLite supports up to 2,147,483,646 tables and indices in a schema, with maximum of 2^{64} (18,446,744,073,709,551,616) rows per table [9]. As a result, after all of these tests, the database was still not corrupted (we can successfully connect to and read the contents). And each query was performed in average of 0.07 second, which was acceptable in our case. Detail of test result is listed in table 1.

TABLE 1. TEST RESULT: 60 THREADS AND 32 QUERIES/THREAD (TOTAL 1920 QUERIES)

Iteration	Number of existing tables	Elapsed time (Real, in second)	Elapsed time (User, in second)	Elapsed time (Sys, in second)
1	0	137.19	1.15	0.36
2	0	137.39	1.15	0.39
3	0	136.55	1.01	0.38
4	60	136.38	1.09	0.42
5	120	140.19	1.17	0.38
6	180	138.90	1.15	0.38

There was one minor note during our experiments. Normally, when we inserted 30 rows of integer value to a newly-created table, we would expect that maximum ROWID value for that table should return 30. But one query returned 29. This was however understandable because we put no delay between table creation, data insertion, and reading. And since multithreading is non-deterministic [10], by default, we have no control which thread will be running after lock to the database has been released. This will also be discussed later.

As a side note. Before we came up with 60 threads and 30 rows number, we tried another combinations. We stop there because this was reasonable enough for our needs. With 60 threads and each running 32 SQL commands (one for table creation, 30 for data insertion, one for reading), we already had 1920 queries sent to the server for each test. We tried more threads and rows, and the result is similar in elapsed time, but more problems are found due to non-deterministic behavior of multithreading. Detail of this test result is listed in table 2.

TABLE 2. TEST RESULT: NUMBER OF THREADS, NUMBER OF QUERIES PER THREAD, ELAPSED TIME, AND NOTE

Iteration	Number of threads	Number of queries per thread	Elapsed time (Real, in second)	Average time per query	Note
1	100	102	760.46	0.075	
2	100	202	1,465.27	0.073	
3	>100	>200	Canceled	N/A	Multithreading related problem, database not affected

Server will response in JSON format (application/json MIME type). This is true for all kind of request. For example, server does not care whether client is requesting for / or /testing path (however, as mentioned earlier, we expect only / path). As long as q (along with optional c variable) is found, then SQL command will be executed and whatever the result is, will be returned. When no q variable is found, an empty object {} will be returned. For curiosity only, we tested SQL command to get version of SQLite database engine (which will not affect the data), and this actually run quite fast. We spawned 1000 to 2000 threads and each query was performed in average of 0.001 second. Detail of this test result is listed in table 3.

TABLE 3. TEST RESULT: NUMBER OF THREADS AND ELAPSED TIME FOR GETTING DATABASE VERSION

Iteration	Number of threads	Elapsed time (Real, in second)	Average time per query
1	1000	1.16	0.001
2	1000	1.11	0.001
3	2000	4.87	0.002
4	2000	2.54	0.001

We also tested erroneous SQL commands: erroneous syntax and call to undefined function. And, thanks to fast SQLite engine, each query was performed, also in average of 0.001 second. We spawned 2000 threads and each thread run 2 erroneous SQL commands. Detail of this test result is listed in table 4.

TABLE 4. TEST RESULT: NUMBER OF THREADS AND ELAPSED TIME FOR ERRONEOUS QUERIES

Iteration	Number of threads	Number of queries per thread	Elapsed time (Real, in second)	Average time per query
1	2000	2	5.02	0.001
2	2000	2	5.10	0.001

There are two types of response: normal and failed SQL operation. The former will result in 'data' key, whose value is the result. And, the latter will result in 'error' key, whose value is the error message, if any. That way, client should first check for the existence of these keys. Screen shot error response from server is shown in figure 3.

```
[Thread-984][response: {'data': [['3.8.7.1']]]]
[Thread-986][response: {'data': [['3.8.7.1']]]]
[Thread-985][response: {'data': [['3.8.7.1']]]]
[Thread-989][response: {'data': [['3.8.7.1']]]]
[Thread-988][response: {'data': [['3.8.7.1']]]]
[Thread-987][response: {'data': [['3.8.7.1']]]]
[Thread-993][response: {'data': [['3.8.7.1']]]]
[Thread-991][response: {'data': [['3.8.7.1']]]]
[Thread-990][response: {'data': [['3.8.7.1']]]]
[Thread-992][response: {'data': [['3.8.7.1']]]]
[Thread-994][response: {'data': [['3.8.7.1']]]]
[Thread-995][response: {'data': [['3.8.7.1']]]]
[Thread-997][response: {'data': [['3.8.7.1']]]]
[Thread-996][response: {'data': [['3.8.7.1']]]]
[Thread-998][response: {'data': [['3.8.7.1']]]]
[Thread-999][response: {'data': [['3.8.7.1']]]]
[Thread-1000][response: {'data': [['3.8.7.1']]]]
[Thread-759][response: {'data': [['3.8.7.1']]]]
[Thread-9][response: {'data': [['3.8.7.1']]]]

real    0m1.115s
user    0m0.852s
sys     0m0.252s
```

Figure 3. Testing with error response from server, with error messages

More testings, with similar results have been performed to verify reliability.

V. CHALLENGES IN CURRENT VERSION

We have at least two challenges in this version of server. The first is blocking SQL operation that force another clients to wait. And the last is we cannot guarantee that commands are executed in order they received.

For the latter, we would like to propose the usage of priority queue. Each request is queued along with its time stamp, and priority queue will pick the smallest one to execute the SQL command. Since time stamp may be in floating point, the smallest number should be easier to determine. Priority queue data structure is commonly found in modern programming language implementations, at least for Java and Python.

For the blocking SQL operation challenge, the main problem is that the server acquires lock, opens the database, executes a query, does a commit when told so, and closes the database. As a comparison, we wrote a test program that simulate our test case, that run locally (direct access to the database file), but without committing change for each operation, and we only commit/close the database at the end of test. And the result was as predicted: 50,002 queries in average of total 0.5 second, or 0.00001 second per query. To be fair, we should not compare between network and local file

operation (using only one thread), but since the server was running at the same host and the result was differ too much, we may assume that synchronization and open/commit/close database operation are the main source of problem. We may change the communication protocol, or using HTTP with session to make it persistent, or we may change how we deal with database operation. But we have no definitive proposal as this writing.

VI. DISCUSSION AND FUTURE DEVELOPMENTS

Currently, one of common features found in client/server database system is authentication and authorization. However, by default, SQLite is serverless database without concept of user account. But, since we propose a server for SQLite database, we should prepare for some way to secure the database. The simplest one would be limit access to the database by connecting host. Before we check the query, we look first for allowed host and reject necessarily. Or, we can use authentication extension [11].

Another possible method would be emulating authentication and authorization feature by using one special table that store user account, hashed password, and access rules. That special table is stored in the database itself, so the database file is portable to another hosts. This method is partially adopted by free/open source software SQLiteBoy [12]. Alternatively, one special database can be used, so user account and related information are not stored in active database. However, then we should provide a mechanism to allow such meta data to be exported if the database file is moved to another host.

We are using HTTP partly because it is ubiquitous. However, we do aware that there are several database-independent and industry-standard method to work with database systems, such as JDBC (for Java platform) and ODBC. Additionally, most client/server database systems, for example PostgreSQL, provide their own protocol [13]. Should we emulate some database protocol? Should we provide a driver for database-independent method?

We have learned that brute force testing procedure makes the server very busy. This can be worse because access to the database is synchronized. On the other side, it is difficult to imagine that our SQLite server must respond to hundred of threads concurrently. After all, this is only a small application, for simple purposes. So, we may implement some resource throttling to prevent too many connections to the server.

Considering the reliability of SQLite database and powerful HTTP/JSON-based database server, proposed method in this paper is planned to be included in Pangsit, a small domain-specific programming language for data entry [14].

VII. CONCLUSION

Looking at the test results, we conclude that proposed method is applicable. During the tests, we could not find any database corruption. And, average of elapsed time for SQL queries is also acceptable.

Certainly, there are some challenges. But, we hope that, by adopting some ideas discussed in future developments, or better ideas from another client/server database systems, we may improve the server, and make it more usable in real world scenarios.

It is interesting to see this SQLite server in action. SQLite, as mentioned earlier, is powerful, fast and reliable database system. Our server is nothing compared to what provided by SQLite. But still, we hope that this server might be useful for someone else, and since it is a free/open source software, we would be happy to see further developments.

REFERENCES

- [1] The SQLite Development Team. About SQLite. [Online]. Available: <https://sqlite.org/about.html>
- [2] The SQLite Development Team. SQLite Frequently Asked Questions. [Online]. Available: <https://sqlite.org/faq.html>
- [3] M. Owens, "Embedding an SQL database with SQLite," *Linux Journal*, 2003(110), 2.
- [4] The SQLite Development Team. Most Widely Deployed SQL Database Engine. [Online]. Available: <https://sqlite.org/mostdeployed.html>
- [5] R. Fielding et al, "Hypertext transfer protocol--HTTP/1.1 (No. RFC 2616)", 1999.
- [6] N. Nursetov et al, "Comparison of JSON and XML Data Interchange Formats: A Case Study. Caine", 2009, 157-162.
- [7] JSON developer. JSON. [Online]. Available: <http://www.json.org>
- [8] R. Fielding and J. Reschke, "Hypertext transfer protocol (HTTP/1.1): Message syntax and routing", 2014.
- [9] The SQLite Development Team. Implementation Limits For SQLite. [Online]. Available: <https://www.sqlite.org/limits.html>
- [10] J. Burnim, and K. Sen, "Asserting and checking determinism for multithreaded programs," In "Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering", 2009, August, pp. 3-12, ACM.
- [11] The SQLite Development Team. SQLite: Documentation. [Online]. Available: <http://www.sqlite.org/src/doc/trunk/ext/userauth/user-auth.txt>
- [12] Noprianto. Simple Web SQLite Manager/Form/Report Application. [Online]. Available: <https://github.com/nopri/sqliteboy>
- [13] The PostgreSQL Global Development Group. PostgreSQL: Documentation: devel: Frontend Backend Protocol. [Online]. Available: <https://www.postgresql.org/docs/devel/static/protocol.html>
- [14] Noprianto and B. Soewito, "Pangsit: spreadsheet-like domain-specific programming language for data entry application development," In "The Eleventh 2016 International Conference on Knowledge, Information and Creativity Support Systems (KICSS)", 2016.

Noprianto is a software engineer in Jakarta, Indonesia. He has been in software industry since 1997. In 2000 to 2003, he was a teaching assistant/research staff at Software Laboratory, Bina Nusantara University, Jakarta. In 2002 to 2011 he was an exclusive contributor for a computer magazine in Indonesia. He has written several published computer books including Python (2002), Debian GNU/Linux (2006), and OpenOffice.org (2006). He also wrote several free electronic books including wxWidgets (2006), Python (2009), SQLiteBoy (2014), and OpenERP (co-author/editor, 2014). Noprianto holds a bachelor degree in Computer Science (2003) and a master degree in Management of Information Systems (2015) from Bina Nusantara University. Currently, he is a doctoral computer science student with research interests in domain-specific programming languages, data entry/reporting, and embedded database. His books and free/open source software can be downloaded from <http://noprianto.com>.

Benfano Soewito completed his bachelor level education in the F MIPA (since April 10, 2008 changed into the Faculty of Science and Technology) department of Physics, Airlangga University, Surabaya, Indonesia. Then he completed his master level education in the field of Computer Engineering at

the Department of Electrical and Computer Engineering, Southern Illinois University, United States of America (USA) in 2004. Finished his doctoral level education in 2009 at the same faculty and university. After completed his undergraduate level education, Benfano worked at a company in Surabaya, which produces crystal oscillator as the head of engineering with the main task in the development of design specification, manufacturing technique and quality improvement of crystal oscillator unit. While completing his master and doctoral degree, Ben also worked as an assistant professor, research assistant, support computer specialist, and developed several websites inside the environment of Southern Illinois University, USA. After finished his doctoral studies, Ben chose to pursue a career in the education field by joining Bakrie University. Since 2013, Ben became full-time lecturer at the Faculty of Graduate Programs at Bina Nusantara University, Jakarta. He has a high interest toward researches in computer science with special interest in information technology as include Internet packet processing and scanning, router development as well as security and computer network.

Sani M. Isa (M'2015) Finished his Ph.D. in Computer Science, University of Indonesia on 2014. Currently he is a researcher and lecturer at Master in Computer Science, Bina Nusantara University, Indonesia. His research interests are data mining, signal/image processing, remote sensing, and biomedical engineering.

Karto Iskandar is a Doctoral in Computer Science student's at BINUS University, Jakarta, Indonesia. He received the Bachelor Degree in Computer Science and Master Degree in Magister Management streaming Information System from BINUS University in 2002 and 2004, respectively. His research interests include Knowledge Management System, Software Engineering, Mobile programming, elderly Education System, digital system, and virtual futures educations. He also served as an IEEE member since 2016.

Dr. Ford Lumban Gaol is currently Associate Professor Informatics Engineering and Information System, Bina Nusantara University. He is the Vice Chair of Bina Nusantara University Doctorate Program in Computer Science and Research Interest Group Leader "Advance System in Computational Intelligence & Knowledge Engineering". Dr. Ford is the Vice Chair of IEEE Indonesia section for International and Professional Activities. Dr. Ford is the ACM Indonesia past Chapter Chair. Dr. Ford is the IIAI Indonesia Chapter Chair. Dr. Ford already involved with some project relate with Technology Alignment in some of multinational companies as well as some government projects. For International highlight, Dr. Ford is the recipient of Visiting Professor in Kazan Federal University, Russia 2014 and 2015, Visiting Professor in Vladimir State University, Russia 2016, Invited Scholar in Aligarh Muslim University, keynote speaker in ICCNT 2014 and Invited Scholar in ICTP Trieste Italy. Dr. Ford is member of Indonesian Mathematical Society (IndoMS), The Association for Computing Machinery (ACM) Professional, The International Association of Engineers (IAENG), and the Indonesia Society for Bioinformatics. He holds the B.Sc. in Mathematics, Master of Computer Science, and the Doctor in Computer Science from the University of Indonesia, Indonesia in 1997, 2001 and 2009, respectively.

Raymond Kosala is a lecturer at the Bina Nusantara University, Jakarta, Indonesia. He obtained his Bachelor degree in Electronics Engineering from the Atma Jaya University, Indonesia (1994), Master of Engineering in Artificial Intelligence (1999), and PhD in Informatics (2003) degrees from the Katholieke Universiteit Leuven, Belgium. His research interests include games technology, data analytics, machine learning algorithms and their applications.