*International Journal of Engineering & Technology*, 8 (1.4) (2019) 448-457

International Journal of Engineering & Technology

*Website: www.sciencepubco.com/index.php/IJET*

***Research paper***

# Techniques for Using Server-side Node.js Modules with the Truffle Ethereum Development Framework

**Hyun-Min Eom[1], Jae-Hwan Jin[2], Myung-Joon Lee[3]***

*[1,2,3]Department of Electrical/Electronic and Computer Engineering, University of Ulsan, 93, Daehak-ro, Nam-gu, Ulsan 44610, Republic of Korea*

**Corresponding author E-mail: mjlee@ulsan.ac.kr*

## Abstract

Truffle is a framework that provides compiling, testing and systematic project management for developing Ethereum decentralized applications. As of now, Truffle provides a way to easily deal with bundling node.js modules of decentralized application using the webpack tool. However, due to the Truffle project structure, server-side node.js modules such as network communication modules are not usable in a direct way. In this paper, to address this issue, we propose a method to use server-side node.js modules through Ethereum smart contracts and event processing mechanism. In the proposed method, a separate node application is associated to the server-side module to execute the module in response to the request of the decentralized application. To this end, we introduce the notion of function gateway, a smart contract for connecting two applications with Ethereum's event-watch processing technique. Also, to use the function gateway contract in a robust way, we introduce a robust function gateway that includes the process of confirming whether or not the event-watch has occurred and the node.js module function has been executed. In addition, we present a decentralized application using node.js module for sending actual e-mails based on the function gateway.

***Keywords***: *Ethereum, Decentralized Application, Truffle, Function Gateway, Blockchain.*

## 1. Introduction

Ethereum[1] is a platform for developing decentralized applications to run on a blockchain[2,3], which is virtually impossible to be forged or tampered. An Ethereum decentralized application has the advantage that stable services can be provided because it stores resources necessary for the application into the associated blockchain network.[4] In addition, the Ethereum decentralized application generally provides a web-based platform-independent user interface using JavaScript, so there is no need for a separate installation process. Based on these characteristics, Ethereum decentralized applications are used in various fields such as games, virtual currency, auction, SNS, electronic voting and so on[5]. To develop Ethereum decentralized application, application developers implement smart contracts with the Solidity programming language[6] and JavaScript-based web applications interacting with the smart contracts.[7] Developing a decentralized application requires the process of compiling these Solidity smart contracts and distributing them to the Ethereum blockchain. Recently, Truffle is introduced as a framework that performs this process with simple instructions and makes it easy to test.

Truffle[8] systematically manages source codes and libraries for Ethereum decentralized applications using a node.js[9] project, enabling connection to the associated Ethereum blockchain network through a simple configuration file. Truffle supports its project configuration based on JavaScript ES6. Recently, Truffle has adopted Webpack[10], a tool for bundling modules to minimize network cost, as the number of node.js modules used in decentralized applications had increased. As a result, developers can use Truffle and Webpack to solve complexity issues among multiple node.js modules and to load all resources such as JavaScript codes, stylesheets, and images as JavaScript modules. However, since Webpack is a client-side module bundling tool for web browsers, it cannot use server-side modules needed for network communication such as email sending or off-blockchain web service. This problem is a major limitation in implementing various functions of decentralized applications using Truffle. So, to provide functions necessary for various situations and to develop decentralized applications in a wide variety of fields, there is a need for an easy and effective solution to such problem.

In this paper, as the extension of our previous work [11], we propose a technique to solve the limitation of the Truffle project by using the Ethereum smart contracts and event processing mechanism. The proposed method enables Ethereum application developers to use the server-side node.js module through the Event-Watch processing technique available to Ethereum smart contracts and decentralized applications. In the proposed method, a separate node application that runs the server-side node.js module is added, and is performed by the request of the decentralized application. To this end, we introduce the concept of function gateway, a smart contract for connecting node.js modules and decentralized applications, presenting the mechanism required for its operation. Also, we introduce a robust function gateway contract that includes the process of confirming whether or not the event-watch has occurred and the node.js module function has been executed in order to use the function gateway contract in a robust way. To be practical, we provide a skeletal function gateway that can be applied to various situations. To show the effectiveness of the proposed technique, we present a decentralized application using node.js module for sending actual e-mail based on the skeletal contract.

This paper is organized as follows. In Section 2, we briefly introduce the Ethereum blockchain, Truffle development framework, and nodemailer module. In Section 3, we propose the event-based function gateway technique, and in Section 4, we describe the development of a decentralized application using the proposed technique. Finally, conclusions of this paper and future directions are given in Section 5.

# 2. Background

## 2.1. Ethereum

The Ethereum blockchain, which emerged as a second-generation block chain, is the platform to support the development of decentralized applications[12]. Unlike Bitcoin, which was the first generation blockchain used mainly for transaction details in blockchain network[13], Ethereum can operate various functions by using smart contracts in the blockchain network. Smart contracts[14] are programmable contracts that have variables and methods that can perform pre-defined functions when they meet certain conditions. Decentralized applications based on Ethereum can provide reliable services to users thanks to the Ethereum blockchain. Decentralized applications are implemented based on HTML and JavaScript in order to work as a web application in connection with the blockchain through the web3.js library. Currently, Ethereum decentralized applications appear in various domains such as games, virtual currency, auction, SNS, electronic voting as shown on the Ethereum website.

## 2.2. Truffle

Truffle is a framework that supports the development and testing of Ethereum decentralized applications. Developers can create a basic source code structure for decentralized applications by using the init command of Tuffle. The created basic structure of decentralized applications includes HTML pages, JavaScript source codes, node.js modules, and smart contract source codes written in solidity language. In addition, to deploy smart contracts, developers can create a simple configuration file for interacting with the Ethereum blockchain network. The Truffle project facilitates management of projects because it categorizes and stores HTML pages, JavaScript source codes, and smart contract source codes for decentralized applications. The process of testing decentralized applications using Truffle can be performed using simple commands in the order of compiling smart contract source codes, deploying smart contracts, and running a Web server to run decentralized applications. Currently, Truffle is undergoing a variety of updates, including the introduction of Webpack, which bundles node.js modules and uses them as a single file in addition to the use of JavaScript ES6. This allows developers to use Truffle to develop decentralized applications that use more complex functions and multiple modules.

## 2.3. Nodemalier

Nodemailer is a node.js module that allows sending of emails by entering simple information[15]. Application developers can provide functions for sending email through importing the nodemailer module into the target node application and creating a transporter object of the SMTP transport. The SMTP transporter object can be created by entering the SMTP server's host, port, and authentication information, and the object can be used for various purposes, such as local SMTP server or Google SMTP. In addition, information such as sender address, receiver address, subject, and text body can be inserted as options for mail. Once the SMTP transporter and email options are created, it is possible to simply send the email through the sendMail () function of the transporter object, and it is also possible to process its response using the registered callback function. The nodemailer module has the limitation of not running on a Web browser because it is a server-side module. Currently, nodemailer supports ES6 JavaScript to solve the memory leak problem, and it is aiming for secure mail transmission through TLS / STARTTLS secure mail transmission and OAuth2 authentication method.

# 3. Event-based function gateway

In this section, we introduce two function gateway techniques to solve the limitations of the Truffle tool for the easy development of Ethereum decentralized applications. To enable Ethereum application developers to use server-side node.js modules in association with the Truffle tool, both of the techniques utilize the event processing mechanism of the Ethereum blockchain.

## 3.1 Function gateway

To use server-side node.js modules in association with the Truffle tool, we introduce the notion of function gateway which utilizes an additional node application and the event-watch processing technique of the Ethereum blockchain. The event-watch processing technique is a process of receiving events from smart contracts in Web3-based Javascripts to perform functions specified by the events. A function gateway technique needs a node application that can invoke server-side node.js modules according to the events received from the associated contract. The application structure using a function gateway is shown in Figure 1. For this, a function gateway contract is added to an existing Truffle project to support the connection of decentralized applications and the new node application. In the added function gateway contract, there is an event for executing server-side modules and a function for firing the event. A decentralized application fires an event by invoking the function of the function gateway contract at the time required. The node application receives the event fired through the function of the function gateway contract and performs the necessary functions according to the events. Since this node application does not use Webpack, it can use server-side node.js modules that are difficult to use in existing Truffle decentralized applications. In the Ethereum blockchain event-watch processing technique, an instance of the contract is needed to receive events fired from the smart contract. So, to create an instance of function gateway registered in the blockchain, we perform the process shown in Figure 2.

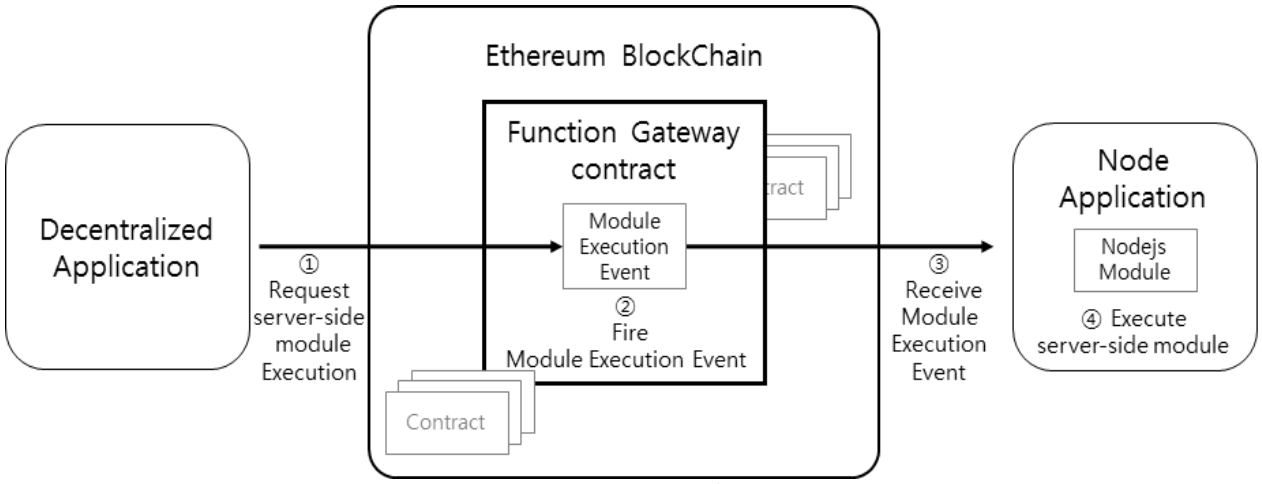


**Fig. 1:** Application structure using function gateway

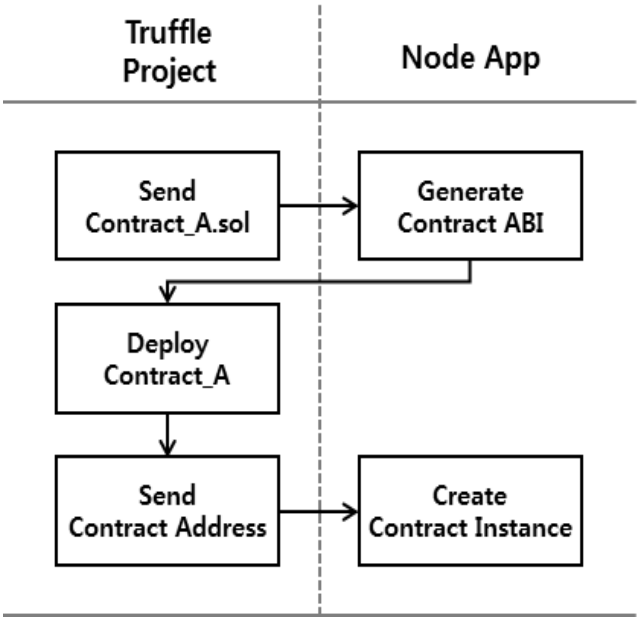


**Fig. 2:** Process for creating contract instance

A developer writes the function gateway contract source code in the Solidity language to connect the decentralized application with the node application. Then, the developer creates the function gateway contract ABI(Application Binary Interface) using the contract source code. Next, the developer deploys the function gateway contract to the Ethereum blockchain using Truffle's migrate command, and passes the returned function gateway contract address to the node application. Lastly, the node application creates an instance of the function gateway contract using the received address of the contract and the generated ABI. The basic code fragment for the event-watch processing in the Ethereum blockchain is shown in Table 1.

**Table 1:** Structure for event-watch

```
... //generate contract ABI
var contract_instance = contract_abi.at(contract_address);
//generate event object that is filter object
var event = contract_instance.Event();

event.watch(function(error, result) {
 // callback function
  if(error) {
    //Post-processing of errors
}
  else {
    //Post-processing of results
  }
});
```

### 3.2 Robust function gateway

Decentralized applications using the above-mentioned basic function gateway technique cannot check whether an event is fired or a module is actually operated. If it is necessary to check whether the module is actually executed according to the purpose of the decentralized application, then the proposed technique does not provide sufficient implementation base. For this, we introduce more robust techniques for functional gateways, whose structure is shown in Figure 3. The event fired from the functions added in the function gateway

contract is received by the decentralized application, and the decentralized application checks the content of the event to confirm whether the event is fired or the function is performed. To this end, a more robust function gateway contract includes functions to fire events for node.js module execution and events for verifying actual event reception and the module execution. Figure 4 shows the process for the robust function gateway technique. The details of each process in Fig. 4 are as follows.

(1) The decentralized application calls the module-execution-event firing function of the function gateway contract to execute the server-side module.
(2) The function gateway contract fires module-execution-event through the called function.

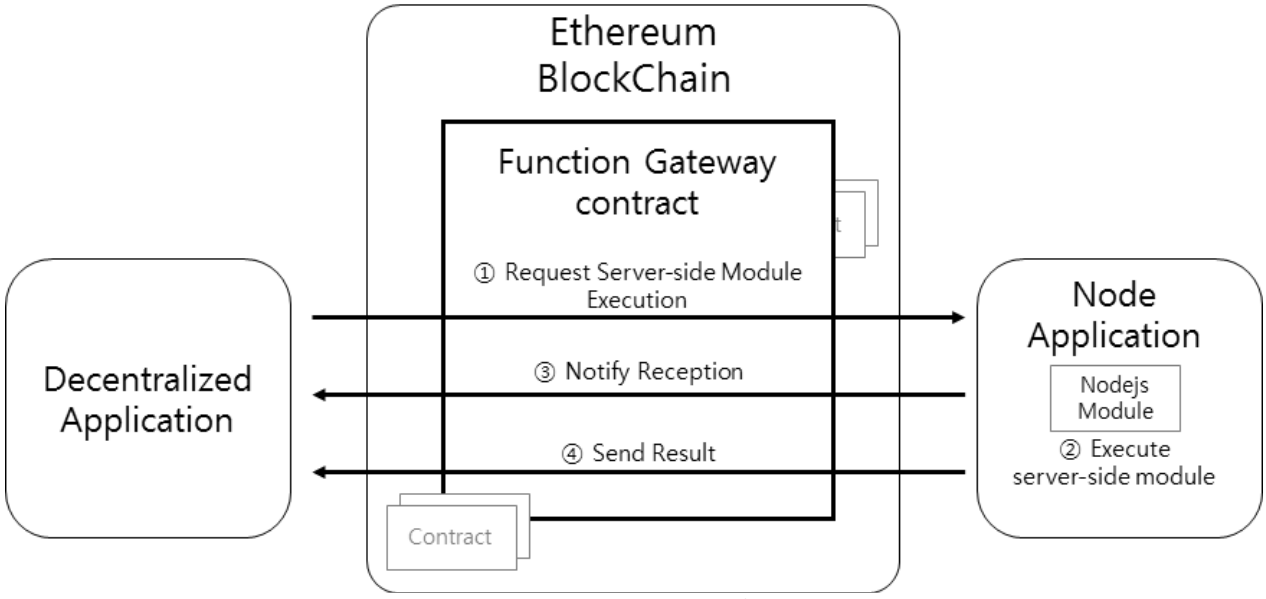


**Fig. 3:** Application structure using robust function gateway

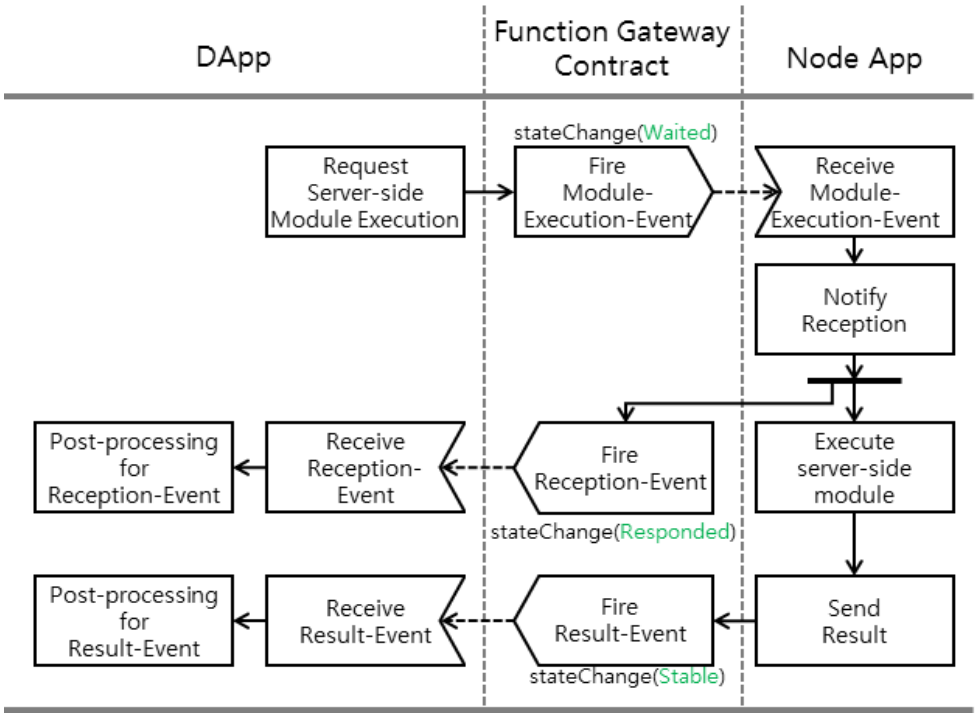


**Fig. 4:** Process of the robust function gateway

(3) The node application receives the module-execution-event and calls the reception-event firing function to respond.
(4) The function gateway contract fires the reception-event through the called function.
(5) The decentralized application receives the reception-event and performs post-processing such as checking whether the event has occurred or not.
(6) The node application calls the result-event firing function of the function gateway contract to deliver the result.
(7) The function gateway contract fires the result-event through the called function.
(8) The decentralized application performs post-processing such as confirming the result-event and receiving the result.

In addition, the robust function gateway contract has state variables to manage its operational state, where each state is defined as Stable, Waited, and Responded. The robust function gateway changes its state according to the progress of the process. Figure 5 shows the state chart for each state of the robust function gateway contract.

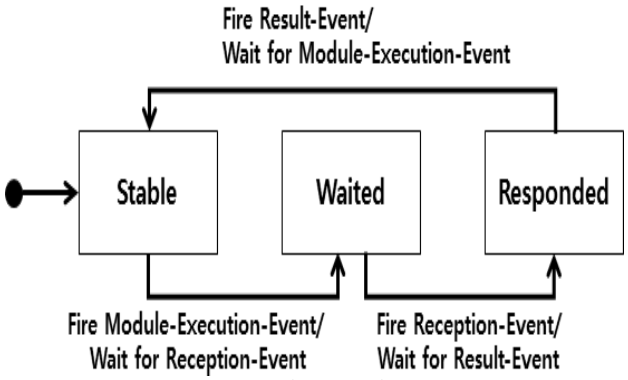


**Fig. 5:** State chart for robust function gateway

The robust function gateway contract is initially in the Stable state. When the event firing function is called from decentralized or node applications, it fires an event and changes its state depending on the function. In addition, to systematically control the operation of the contract, the contract attaches a special modifier to each of its functions, which is an execution condition based on each operation state. Modifiers onlyStable, onlyWaited, and onlyResponded are execution conditions that define a specific function to be executed only in the associated state.

The robust function gateway technique requires additional operational processes to the basic function gateway technique. Surely, the robust function gateway can handle events more reliably, whereas the basic function gateway can processes events more rapidly. In applications where the success or failure of executing a server-side module is very crucial, the robust function gateway technique should be adopted. Ethereum decentralized application developers can implement features that use server-side modules by selecting one of these techniques that meet the needs of the applications they develop.

## 4. Development of decentralized application using function gateway technique

This section describes the development of decentralized application using the proposed function gateway technique. We propose a skeleton contract of the function gateway and describe the implementation of an Email sending service using it.

### 4.1 Skeleton Contract for function gateway

The skeleton contract of the function gateway consists of a module-execution-event and the associated function that fires the event. The skeleton contract for the robust function gateway has additional functions to fire the reception-event and the result-event to confirm the actual event firing and the module execution. It also has state variables to manage the operation state. In addition, the functions of the skeleton contract of the robust function gateway have modifiers described in the previous section. Using the proposed skeleton contract, the developer can create a function gateway contract suitable for the situation with the appropriate parameters. With the help of the created function gateway contract decentralized applications can use the server-side modules by passing the appropriate parameters according to the purpose.

### 4.2 Development of email sending service

Since the e-mail module is a server-side module that uses network communication, it is difficult to use it in the webpack-based Truffle application. In this section, we describe the email sending service developed by applying the proposed function gateway technique. The Email function gateway contract is implemented based on the skeleton contract of the function gateway. The developed decentralized application encrypts email contents, and stores the encrypted email contents into the EmailStore contract. Figure 6 shows the structure of the developed email sending service. The developed decentralized application works according to the following steps.

**Table. 2:** Skeleton of function gateway Contract

| Skeleton contract for function gateway | Skeleton contract for robust function gateway, |
|---|---|
| contract FunctionGateway {<br>event Execution_Event( );<br><br>// constructor<br>function FunctionGateway( ) { }<br><br>//Module-Execution-Event Firing Function<br>function excutionFunction( ) {<br>Execution_Event( );<br>}<br>} | contract RobustFunctionGateway {<br>enum State { stable, waited, responded }<br>State state;<br><br>event Execution_Event( );<br>event Response_Event( );<br>event Result_Event( );<br><br>… //Declare Modifier<br><br>// constructor<br>function RobustFunctionGateway( ) { }<br><br>//Module-Execution-Event Firing Function<br>function excutionFunction( ) onlyStable( ) {<br>Execution_Event( );<br>state = State.waited;<br>}<br>// Reception-Event Firing Function |

```
    function responseFunction( ) onlyWaited( ) {
        Response_Event( );
        state = State.responded;
        }
    // Result-Event Firing Function
    function resultFunction( ) onlyResponded( ) {
        Result_Event( );
        state = State.stable;
        }
}
```

(1) The decentralized application encrypts the email content with the public key of the email server Ethereum account and generates a hash key of the email content.
(2) The decentralized application calls the regEmailData() function of the EmailStore contract to store the encrypted email content.
(3) The EmailStore contract stores the hash key and encrypted email content sent as parameters to the regEmailData () function as key-value pairs in the email information list.
(4) The decentralized application calls the startEmailSending() function of the Email function gateway contract to fire the associated email sending event.
(5) The Email function gateway contract fires the email sending event with a hash key through the startEmailSeding() function.
(6) The node application, which is an email server application, receives the email sending event and checks the hash key, which is a parameter of the event.
(7) The node application calls the getEmailData() function of the EmailStore contract to retrieve the encrypted email information.
(8) The Email Store Contract retrieves the encrypted email data from the email information list using the hash key through the getEmailData() function, delivering it to the node application.
(9) The node application decrypts the received encrypted email content with the private key of the email server account and sends the email using the sendMail() function of the nodemailer module.
But, in many cases, it is necessary to confirm the result of execution of the e-mail module, which means applications should use the robust function gateway technique. Figure 7 shows the structure of the email function gateway contract to implement robust email sending service based on the robust function gateway technique.

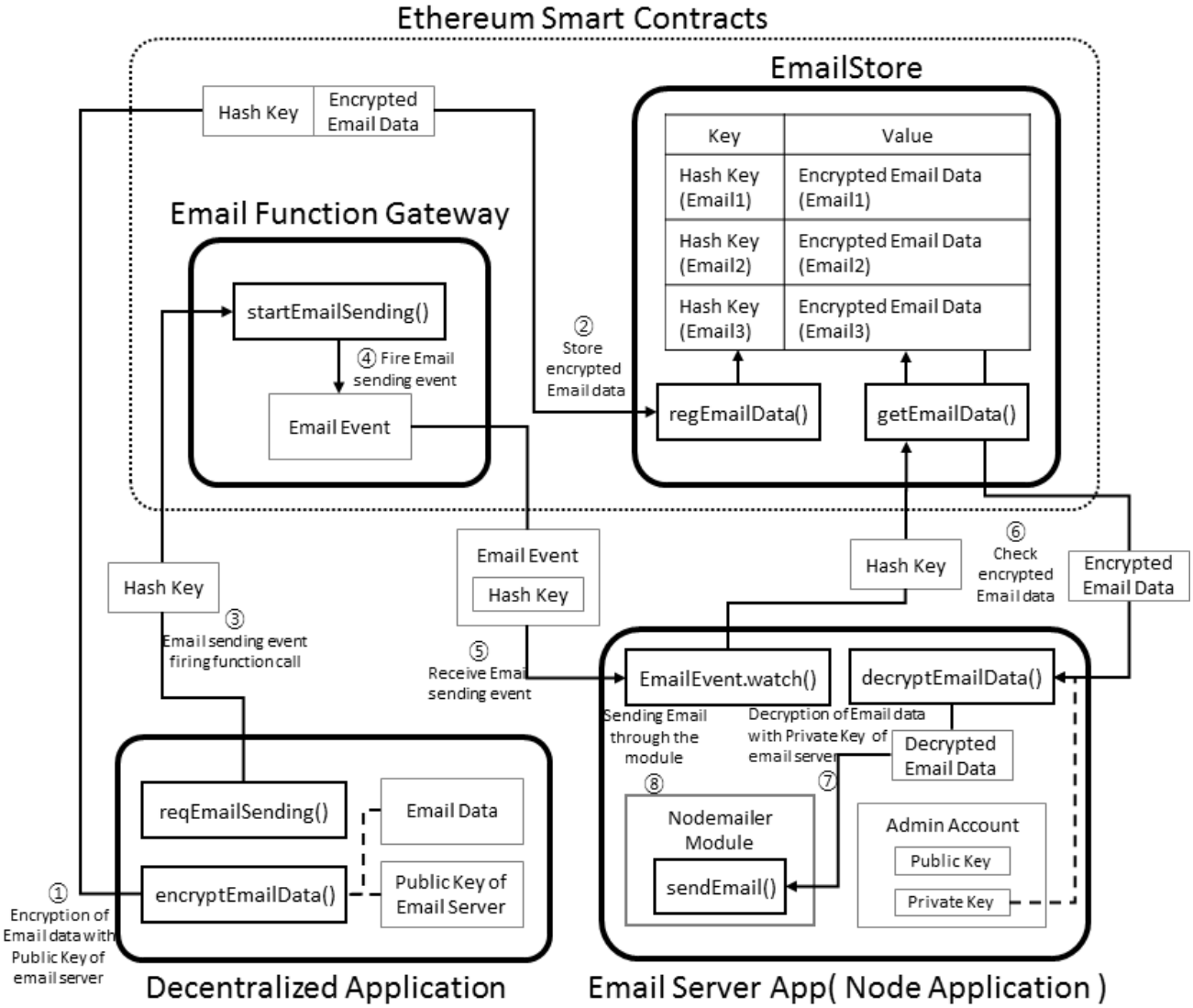


**Fig. 6:** Structure of email sending service

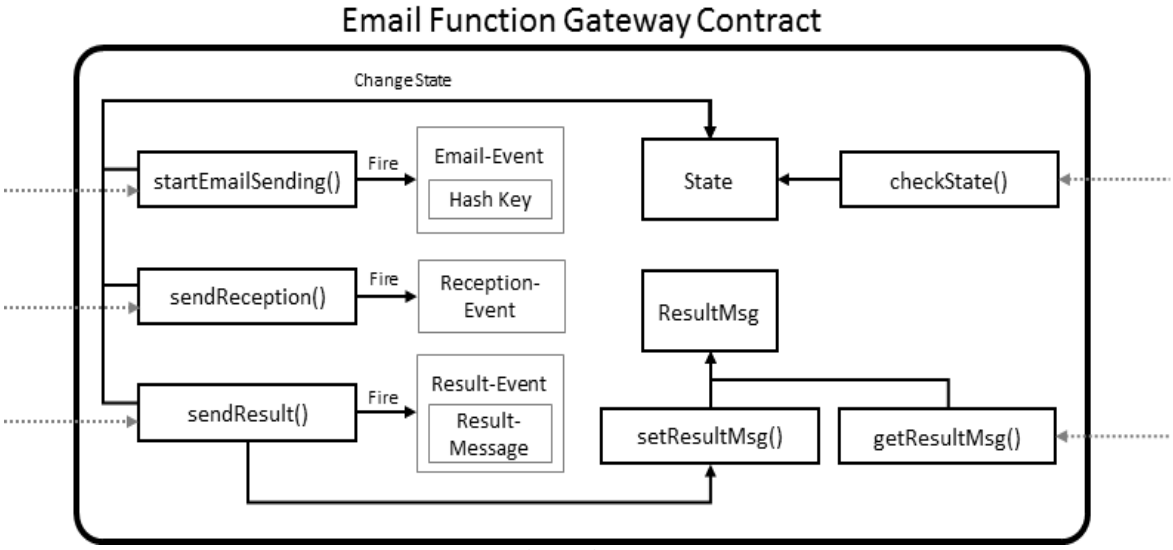


**Fig. 7:** Structure of email function gateway contract

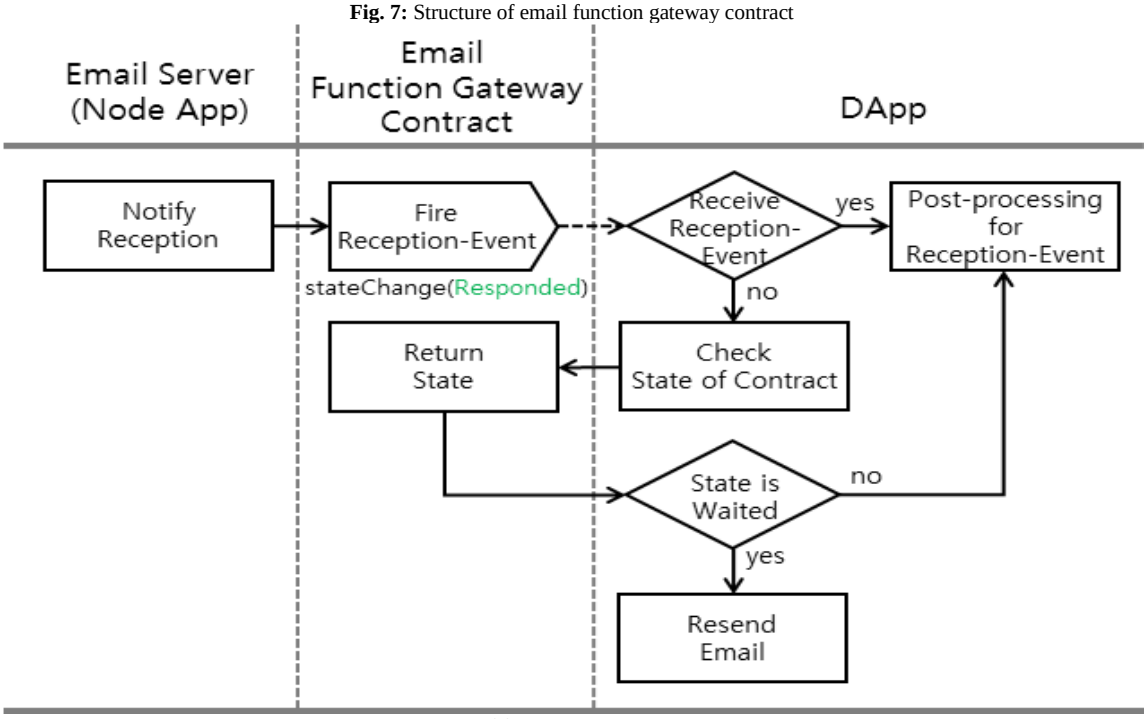


**Fig. 8:** Process of failure to receive a reception-event

A robust e-mail sending service can check the event reception or email sending failure. It can also perform the related recovery steps such as resending email. Figure 8 shows the process of failure to receive a reception-event in the robust email sending service. If the decentralized application does not receive the reception-event within a certain period of time, it checks the status of the Email function gateway contract. If the status of the Email function gateway is Waited, the decentralized application determines that the Email server application has not received the email-sending-event and performs retransmission of the email as the related recovery process. Similarly, when it fails to receive the result-event, the robust email sending service will perform the process as shown in Figure 9.

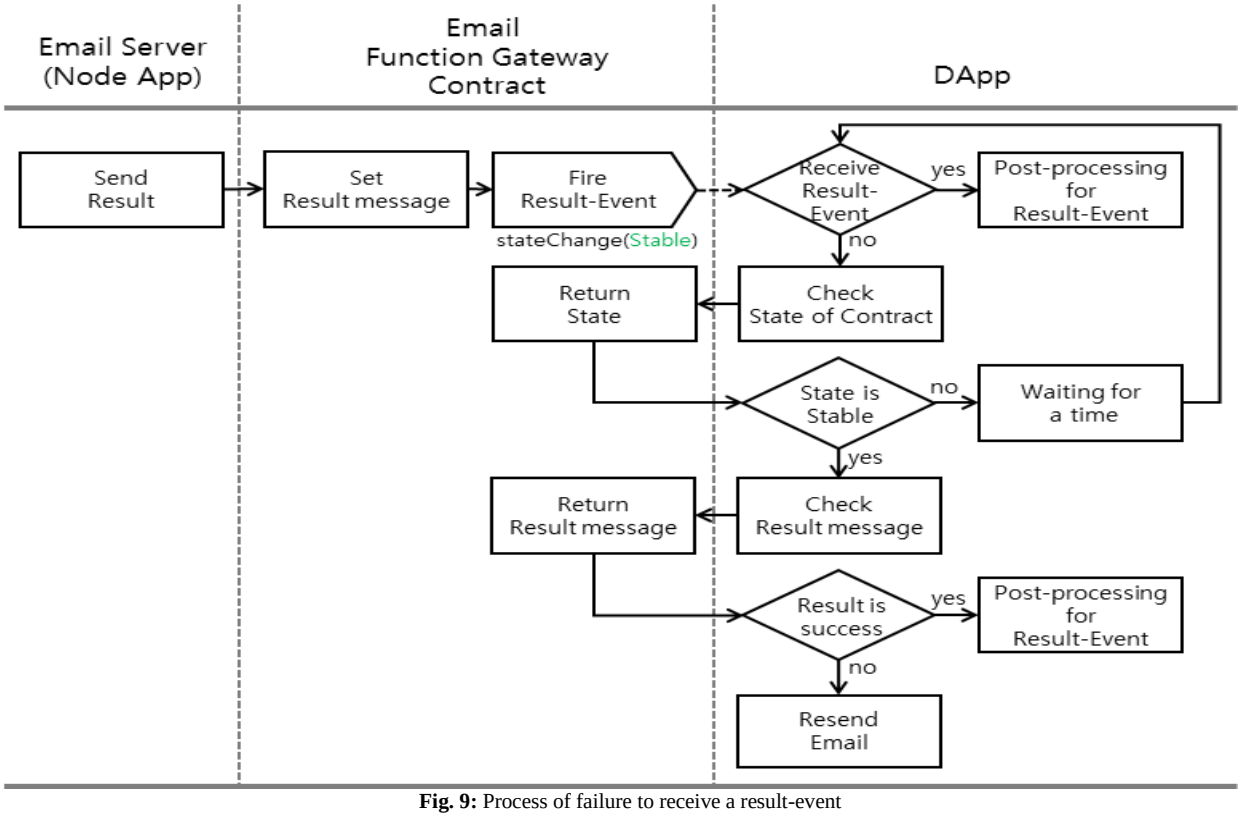


**Fig. 9:** Process of failure to receive a result-event

The details of each process in Fig. 9 are as follows.

(1) The Email Server application calls the sendResult() function of the Email function gateway contract to send the decentralized application the result that returned after the email has been sent.

(2) The Email function gateway contract stores the result received as a parameter of the sendResult() function into the ResultMsg, a variable to check the result if the decentralized application fails to receive a Result-Event.

(3) Also, the sendResult() function of the Email function gateway contract fires a Result-Event containing the result.

(4) If the decentralized application does not receive the result-event within a certain period of time, it checks the current status of the Email function gateway contract.

(5) If the status of the Email function gateway contract is not in the Stable state, the distributed application determines that the email sending process is not completed in the Email server application and waits for a certain period of time again.

(6) If the status of the Email function gateway is in the Stable state, the decentralized application notices that the reception of the result event has failed in some unknown reason, and directly get the result through the getResultMsg() function of the Email function gateway contract.

(7) The decentralized application can then perform post-processing based on the result through the contents of ResultMsg.

### 4.3 Screens of the email sending service

The decentralized application of the email sending service is running on a Web browser. The user accesses the web page for composing and sending email and inputs the recipient, title, and contents of the email. Figure 10 shows a screen for creating the contents of the email to be sent.

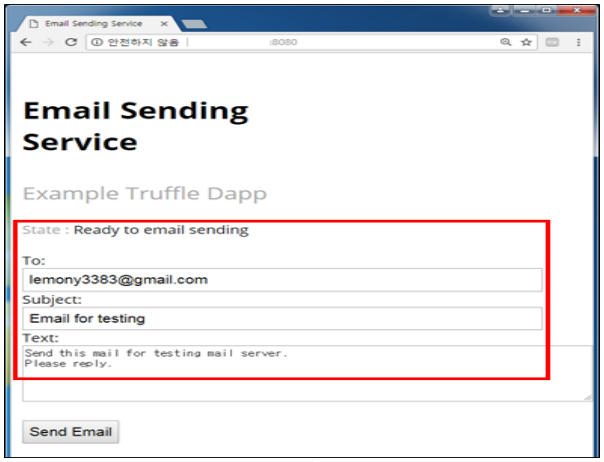


**Fig. 10:** Screen of email composing

After creating the content, the user presses the send button at the bottom of the screen to request the email sending. Then, user can check the email sending process through the status message displayed at the center of the screen. Figure 11 shows status message from each state for the process of sending email.

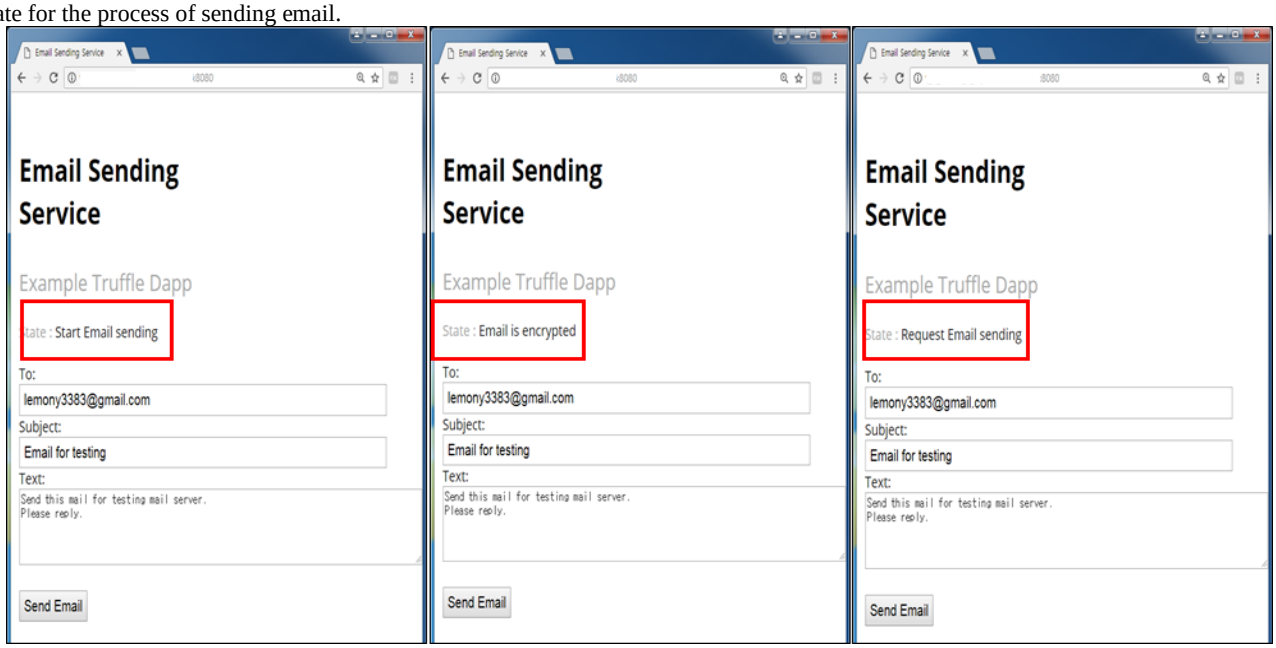


**Fig. 11:** Screens of email sending

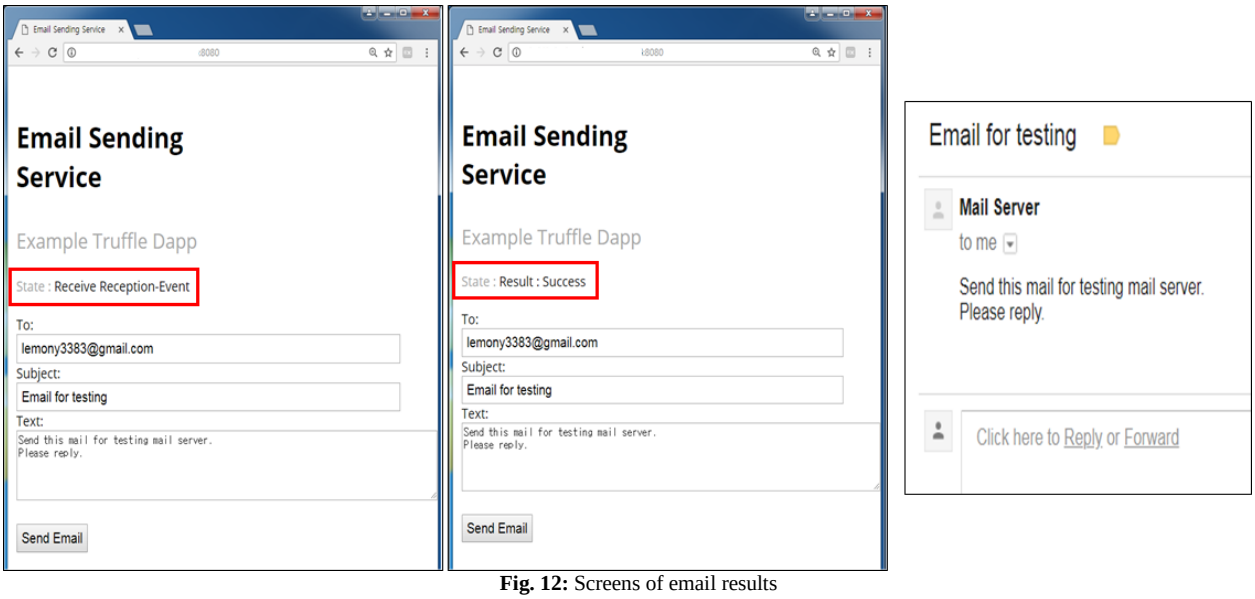


**Fig. 12:** Screens of email results

The email content created by the user is encrypted and requested to be sent through the startEmailSending () function of the Email function gateway contract. In the process of sending email, the status messages "Start Email Sending", "Email is encrypted", "Request Email sending", "Receive Reception-Event", and "Result : (result)" are printed respectively on the decentralized application screen. When all of the processes are completed normally, the status message is changed to the initial value "Ready to Email sending". Figure 12 shows the screen at the time when the email sending is complete and the screen at the time when the email server(like gmail) actually received the email.

## 5. Conclusion

In this paper, we introduced methods for using server-side node.js modules in association with a decentralized application developed by the Truffle and Webpack tools. Thanks to the methods, Ethereum application developers can connect the node.js application that runs the server-side node.js module and the Truffle decentralized application through the function gateway. The function gateway is an Ethereum smart contract to perform the necessary functions in various ways by connecting two applications through the Ethereum's event-watch processing technique.

In addition, we presented an advanced function gateway that includes the process of confirming whether the event-watch is actually occurred and the node.js module function has been executed using the function gateway in a robust manner even in case of insecure event watching environment. Also, to show an actual function gateway contract and practical application, we have developed an email sending service and the associated function gateway contract in association with the Ethereum blockchain, which explains the effectiveness of the

introduced methods by itself. We believe that the methods can be usefully adopted for utilizing multiple server-side modules to develop more feature-rich Ethereum decentralized applications in various application domains.

## Acknowledgement

This work was supported by the 2018 Research Fund of University of Ulsan.

## References

[1] W. Gavin, “Ethereum: A secure decentralised generalised transaction ledger”, Ethereum Project Yellow Paper, Vol. 151, (2014)
[2] A. Kosba, A. Miller, E. shi, and Z. Wen, Hawk: The blockchain model of cryptography and privacy-preserving smart contracts, Security and Privacy (SP), 2016 IEEE Symposium on. IEEE (2016), pp.839-858.
[3] G. Zyskind et al., "Decentralizing privacy: Using blockchain to protect personal data", Security and Privacy Workshops (SPW), (2015) IEEE, pp. 180-184.
[4] Ethereum Dapps, http://dapps.ethercasts.com/
[5] J. H. Jin, H. M. Eom, J. E. Sun, and M. J. Lee, "A Blockchain-based Public Opinion Polling Solution Supporting Robust Verification", In: Proceedings of Convergence Research Letter (2017), Vol. 3, No. 3, pp. 245-248.
[6] Solidity, https://solidity.readthedocs.io/
[7] A. Bogner, M. Chanson and A. Meeuw, "A decentralised sharing app running a smart contract on the ethereum blockchain.", Proceedings of the 6th International Conference on the Internet of Things, (2016), pp. 177-178.
[8] Truffle, https:// github.com/trufflesuite/truffle/.
[9] Node.js, https://Node.js.org/.
[10] Webpack, https://webpack.js.org/.
[11] H. M. Eom, J. H. Jin, and M. J. Lee, "A Technique for Sending Email in association with the Truffle Ethereum Development Framework", International Journal of Private Cloud Computing Environment and Management (2018), Vol. 5, No. 1, pp. 1-6
[12] V. Buterin, “A next-generation smart contract and decentralized application platform”, Ethereum project white paper, (2014).
[13] H. Watanabe, S. Fujimura, A. Nakadaira, Y. Miyazaki, A. Akutsu, and J. Kishigami, "Blockchain contract: Securing a blockchain applied to smart contracts", 2016 IEEE International Conference on Consumer Electronics (ICCE), (2016), pp. 467–468.
[14] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system", (2008)
[15] Nodemailer, https://nodemailer.com/