Regular Paper

GNU Radio-based Cloud Development Environment for Software-defined Radio Users

Hirotaka Suzuki[1,a) Haruhisa Ichikawa[1] Jin Mitsugi[2] Yuusuke Kawakita[3]

Received: March 29, 2018, Accepted: December 4, 2018

Abstract: Software-defined radio (SDR) is used for R&D such as cognitive radio. Because sampling devices and personal computers configuring the SDR environment have fixed configurations, some reconfiguration is needed when the SDR application requires different data sources and computational resources. To enable reconfiguration, we present a cloud platform that has scalable computing resources and data sources deployed over a wide area. We use the existing SDR development environment (SDRDE) and implement it on the cloud platform. It is necessary to transfer the SDR environment to the cloud by separating the UI and data management from the existing SDRDE. In this study, we selected the GNU Radio Companion (GRC) as the base platform and implemented an SDRDE for an unspecified number of users by separating the UI. In addition, we used task parallel and distributed computing for the SDR application. In this study, we focus on compatibility with the base implementation and lifting the limits of computational resources. We confirmed the compatibility with GRC in terms of user skill sets and software assets and evaluated the system response time. Further, the relationship between CPU utilization and instructions per cycle during SDR application execution shows that in general, this approach is effective.

Keywords: software-defined radio, GNU Radio, development environment, cloud computing, task parallel and distributed computing

1. Introduction

Software-defined radio (SDR) is a technology that enables the components of a radio system implemented in hardware to be implemented in software on a personal computer (PC) or an embedded system. By modifying software on a single platform such as a PC, the radio functions for various frequencies and communication systems can be easily changed [1]. SDR is used for research and development tasks such as researching efficient utilization of radio resources in cognitive radio and attempts to implement existing or new communication methods/standards [2]. **Figure 1** (a) shows the general configuration of an environment that utilizes SDR. This environment is generally configured using a sampling device with an analog-to-digital converter (ADC) as a data source, and a PC with an SDR development environment (SDRDE) such as LabVIEW, GNU Radio Companion, or MATLAB + Simulink is connected to the system.

Hardware configurations for the sampling device data source and PC computational resource are fixed. Parameters for the data source include the bandwidth, center frequency, sampling rate, and location of sampling and are subject to the constraints of the hardware and its location. The performance of the SDR application (SDR App) is subject to the constraints of the PC's computa-

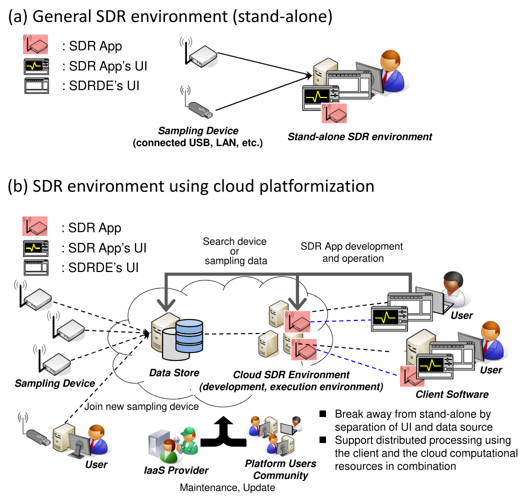

(a) General SDR environment (stand-alone)
: SDR App
: SDR App's UI
: SDRDE's UI
Sampling Device (connected USB, LAN, etc.)
Stand-alone SDR environment
(b) SDR environment using cloud platformization
: SDR App
: SDR App's UI
: SDRDE's UI
Search device or sampling data
SDR App development and operation
User
User
Sampling Device
Data Store
Cloud SDR Environment (development, execution environment)
Client Software
Join new sampling device
User
IaaS Provider
Platform Users Community
Maintenance, Update
Break away from stand-alone by separation of UI and data source
Support distributed processing using the client and the cloud computational resources in combination

Fig. 1 (a) Configuration of a general SDR environment, and (b) configuration of an SDR environment using cloud computing.

tional resources because it runs as a stand-alone program. When the SDR system is preconfigured for a specified objective and use, the configuration of the data source and the computing resources needed to operate the SDR can be estimated. For personal use or research and development, estimating and reconstructing data sources and computational resources in advance is difficult, because the system configuration is not defined according to the change in requirements such as the radio standards/methods and usage data. It is not realistic to reconfigure the SDR execution environment each time the SDR App is reconfigured or to meet temporary performance and requirements. This reconfiguration would diminish the efficiency of research or product and service

[1] The Graduate School of Informatics and Engineering, the University of Electro-Communications, Chofu, Tokyo 182–8585, Japan
[2] The Faculty of Environment and Information Studies, Keio University, Fujisawa, Kanagawa 252–0882, Japan
[3] Faculty of Information Technology, Kanagawa Institute of Technology Atsugi, Kanagawa 243–0292, Japan
a) suzuki.hirotaka@kwkt-lab.org

development.

We propose a platform in which data source and computational resources limits are avoidable by using a cloud-based platform of the SDR environment, as shown in Fig. 1 (b). By separating the operation and management of data sources and major SDR execution environments from the user side, the platform enables the data source to be expanded over a wide area, and the SDR can run using scalable computational resources. As a result, the user can focus on research and development using SDR and actual sampling data for tasks such as experimental development and data analysis. In this study, we also consider parallel and distributed computing (PDC) to increase scalability [3].

Cloud-RAN [4] is an architecture for a mobile network configured using SDR technology and cloud computing. It separates the platform and provides services from the mobile phone carrier. However, in contrast to the requests of Cloud-RAN in terms of a platform for an unspecified number of users, the platform proposed in this paper allows for unspecified services. The unspecified services are not just protocol related; they are also considered in applications such as radio astronomy and the exploratory analysis of radio space. Users can configure and execute any SDR App and access it from one operating environment with multiple data sources.

The ultimate goal of our study is to achieve a cloud platform for SDR using an existing SDR environment. We focus on constructing on the cloud an SDR development and execution environment based on the existing components. We selected an existing SDRDE and modified it for implementation on a cloud platform. There are several challenges when performing this modification using existing stand-alone SDRDEs. First, to enable the user of various data sources, we separate the data-management mechanism from an existing SDRDE. Thus, it is possible to search and reference the sampling device and the stored sampling data. Second, to provide this configuration and execution in the cloud to an unspecified number of users, the UI is separated from an existing SDRDE, and the development and execution environment is relocated to the cloud. In this paper, we deal with the second challenge: developing the cloud development and execution environment and its client software from an existing SDEDE. Regarding the execution environment, we will focus on realizing the split execution of the SDR App with UI separation. The resource management technique when multiple SDR Apps are concurrently executed by an unspecified number of users is the next step and is not handled in this paper.

The implementation presents two challenges. First, compatibility with the original SDRDE's software assets and user skill sets must be maintained. Here, the skill set refers to the skills needed to operate and manage workflows of the existing software. It is desirable for existing users to be able to use their original skills on the new platform. Moreover, users should be able to use the new platform without needing new knowledge. Software assets are elements that are configured or used, such as files, modules, basic data structures, and basic processing mechanisms. Modules and files used in the cloud-based SDRDE must be compatible with those in the existing SDRDE. Second, to avoid limits on computational resources, we realize an execution environment that does not depend on a user's local computational resources. The reuse of software resources, especially UI components, enables task-level PDC of the SDR App using client and cloud computational resources. We focus on CPU and memory computational resources. Additionally, we consider the performance when the SDR App is executed while avoiding constraints.

The rest of this manuscript is organized as follows. The base SDRDE implementation is introduced in Section 2, which also includes related work on the separation of a UI from a stand-alone environment and avoiding computational resource constraints during SDR App execution. The design, implementation, and functional requirements for the proposed implementation are introduced in Section 3. The evaluation is presented in Section 4. Finally, the conclusions are presented in Section 5. This paper is an extended version of a paper presented at APCC2016 [5]. Additional related work, a system response time evaluation, and a more detailed distributed processing evaluation have been added to the original text.

2. Related Work

2.1 Overview of GNU Radio and GNU Radio Companion

We selected the GNU Radio (GR) and GNU Radio Companion (GRC) as the base platform for the SDR environment in this study. GR and GRC has a transparent implementation because it is open-source software with its own development community. In addition, it has been used in several SDR studies, for example, in the studies on spectrum sensing [6], [7] and the avoidance of signal interference [8], [9].

GR is an open-source development toolkit for SDR. GR contains various modules in units called blocks that can be grouped, making it possible to use third-party modules, called out-of-tree (OOT) modules, as well. A graph that describes the signal processing flow of data from the source to the sink is called a flowgraph. The nodes of this graph are the blocks, which are usually written in C++. The blocks written in C++ are available in Python through the Swig interface. GRC is an environment that generates an intuitive block diagram of a flowgraph from GR. The software structure of GR deals with abstraction using a block-definition file and a ".grc" file. The block-definition file presents the information in a block in the form of an XML file. The .grc file is also in XML format, and it can be saved or loaded to structure the information in the flowgraph. GRC can generate Python code, including GR blocks, using a template engine called Cheetah. In this paper, we refer to the block diagram model on the GRC and the structure inside the GRC as a flowgraph, and refer to the Python code that is generated based on the flowgraph as the SDR App.

2.2 Previous UI Separation Studies

CORNET3D [10] is a web application for researching testbeds for dynamic spectrum access (DSA) for wireless communication research and education. It uses a testbed called CORNET [11] with an SDR. CORNET is composed of clusters that can be configured with multiple SDR nodes located within a building. They connect the sampling device of the LAN to the computer. CORNET3D uses the GR in CORNET to plot the spectral data processed by the spectrum sensor application. The spectrum sensing parameter can be adjusted in the web UI. However, parameters cannot be updated by the user during SDR App execution. All communication, such as spectral data and parameter adjustment, is done via WebSocket communication (described below). The functions of the SDR are predetermined for spectrum sensing. Further, the remote functionality does not include changing the configuration of the SDR App. Considering the purpose of CORNET3D, however, the implemented features are sufficient.

Although it is not directly related to UI separation, remote SDR execution has been proposed by the GRC Working Group [12] (GRCWG), who also described the work that is needed for GUI integration. Specifically, remote SDR execution is presumed to refer to deploying SDR App remotely, executing and controlling it over GRC, and providing the GUI of a remotely executing SDR App to users.

In Google Summer of Code 2017, gr-bokehgui, a web-based display mechanism for GNU radio flowgraphs was developed [13]. It provides a web-based plot environment and an interactive environment using the SDR App instead of the Qt framework based GUI of GR. This enables the constraints of a fixed GUI environment for GR's SDR App running on a local system to be avoided. Gr-bokehgui uses the WebSocket protocol to exchange data between the SDR App and the web UI. It also does support dynamic parameter update during the SDR App execution. It can be considered one solution to the GRCWG's proposal.

2.3 Attempts at Distributed Processing

In Ref. [3], it is noted that conflicts in the memory bandwidth of the SDR system caused by shared memory limit scalability. Therefore, [3] proposed a data PDC framework, called GR-Router, that enables distributed processing across multiple computers for the GR flowgraph in the block unit. Blocks executed in parallel are connected via TCP.

2.4 Proposed Concept

Although CORNET3D [10] and gr-bokehgui [13] succeeded in separating the UI from the SDR App running on the local system, for reconfiguring a remote SDR App, it is required to log in to a remote host or to deploy a reconfigured SDR. In this study, our implementation manages the deployment and execution of the SDR App remotely by implementing a usable development environment on the cloud. This is achieved by separating the UI from the standalone environment of GRC. It also provides dynamic parameter update functionality for the cloud-based SDR App just like gr-bokehgui [13]. One form of implementation is described for the remote SDR execution proposed in GRCWG. Gr-router [3] focuses on simultaneously avoiding memory bandwidth constraints and improving processing performance. We introduce task PDC of the SDR App between the cloud and client, allowing the client PC to execute the SDR App without depending on its own computational resource. We are dealing with the avoidance of single host's GPP performance constraints such as clock frequency and number of cores by using the cloud computational resources, and although we do not focus on improvement of processing performance by task-level parallelization, we are considering this possibility. Task PDC can partially alleviate memory bandwidth constraints. Because gr-bokehgui and gr-router are targeted at GNU Radio, they have compatibility with implementation and can be incorporated into the system proposed in this study.

3. System Design and Implementation

Taking the software structure of the GRC and the challenges in this work into account, we describe the design and implementation of the proposed system after defining the function requirements. To separate the UI, all functions except for the UI of the GRC must be deployed on the remote host in a development environment. For the function that generates the SDR App, we add split generation functions for the SDR App to the cloud-side development environment and the user-side client considering UI separation. This function includes the communication function between both generated SDR Apps.

3.1 Functional Requirements

This section describes the four functional requirements for SDR App configuration and SDRDE needed for the UI separation.

3.1.1 API for the UI Functions

We separate the UI from GRC and replace it with a UI operation API to move the SDR configuration and operation functions to the cloud side. The API provides remotely accessible functions available on the GRC UI such as flowgraph editing and SDR App execution management. In the client software, the API function corresponding to the user's operation is called.

3.1.2 Communication Functions of Data and Instructions

In this implementation, the user skills needed to operate the program and software assets should remain as similar as possible. When the cloud-based SDR App must perform real-time output or plotting, it is necessary to send the data from the cloud to the client side and output the data there. To reuse the previous code, we use the existing GR GUI widget for plotting on the client side. We also reuse the GUI widgets (sliders, choosers, check boxes, and text boxes) of the existing GR to change the parameters of the current SDR App. To do this, the client side must have the execution environment of the GR. The client's PC can be regarded as part of the SDR App processing resources. As mentioned in Section 3.1.1, client software is needed for the function to call the API in the cloud. When running an SDR App that needs plots, the cloud side has a remote development environment and runs part of the SDR App, and the client side has the client software and runs the rest of the SDR App. Therefore, it is necessary to implement three communication functions. The first function sends the data that will be processed by the SDR App from the cloud to the client, i.e., in-processing-data communication. The second function changes the parameters of the cloud-based SDR App using the GUI widget of the client-side SDR App, i.e., parameter-update communication. The third function is an API between the client software and the development and execution environment on the cloud, i.e., API communication.

3.1.3 Selecting Block Execution Location

As mentioned in Section 3.1.2, the client's PC can be regarded as part of the processing resources for the SDR App. When using the GUI widget, the client PC inevitably becomes a partial execution host for the SDR App. Hence, we introduce task PDC between the client and cloud. Task PDC is used because the blocks constituting a flowgraph are executed as an SDR App using a thread-per-block scheduler. The flowgraph is divided using blocks as the units, that is, the SDR App is divided into units of threads and rearranged to multiple hosts. Setting the execution location for each block enables the computational resource consumption of the client's PC to be configured flexibly by enabling some resources to be consumed in the task PDC of the SDR App. It is possible to mitigate the cost of the computational resources on the cloud by using a reasonable amount of computational resources of the client's PC.

Introducing the Task PDC to the SDR App has two requirements. First, the dependency relationships among the blocks in the flowgraph before split must be maintained in the distributed SDR App. Second, setting and identifying the split positions of the flowgraphs to be split must be possible. With regard to the first requirement, among the blocks constituting the flowgraph, some blocks are responsible for the functions of parameters and variables, which are referenced from one or more blocks, and have dependencies. Handling this parameter block is a hurdle in splitting the flowgraph and realizing the Task PDC. We must have the function of synchronizing the parameters and the function of identifying the parameter block having the dependency. The former has already been mentioned as a parameter-update communication function in Section 3.1.2. For the second case, when designating which cloud side or client PC side to execute on each block constituting the flow is specified, a function capable of identifying the boundary of split is required. When specifying the execution location for each block, it is necessary to be able to identify in advance the block on which the execution on the client PC side having the GUI environment is a prerequisite like the block having the GUI plot function.

3.1.4 SDR App Split Generation by Flowgraph Split

Flowgraph split is processed using the execution location settings of each block. It is hence necessary to generate an SDR App that contains the essential functions for communication. The communication functions to be included in the split-generated SDR App are the in-processing-data communication and the parameter-update communication mentioned in Section 3.1.2. However, it is not preferable to force the user to add this function during flowgraph editing. The communication function between the split SDR Apps should always be automatically incorporated so that it does not need to be consciously added from the flowgraph side.

3.2 System Design and Implementation

This section describes the design and the implementation of each functional requirement. **Figure 2** shows an overview of the entire implementation. We changed GNU Radio v3.7.9's GRC based on functional requirements, and we call the cloud side the "Server GRC" and the client side the "Client GRC." We developed the proposed system using Python 2.7, which was also used to implement GRC. The Server GRC provides an API through

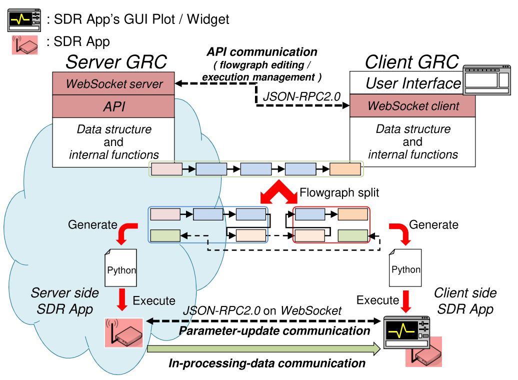

: SDR App's GUI Plot / Widget
: SDR App
Server GRC
WebSocket server
API
Data structure and internal functions
API communication (flowgraph editing / execution management)
JSON-RPC2.0
Client GRC
User Interface
WebSocket client
Data structure and internal functions
Flowgraph split
Generate
Generate
Python
Python
Server side SDR App
Execute
Execute
Client side SDR App
JSON-RPC2.0 on WebSocket
Parameter-update communication
In-processing-data communication

Fig. 2 Implementation overview of the Server and Client GRCs.

WebSocket by replacing the GUI function. Flowgraphs being edited are shared between the Server GRC and Client GRC. The flowgraph is split between the Server GRC and Client GRC when Python code is generated during SDR App execution. Parameter changes due to the operation of the GUI widget for the flowgraph running on the client side are reflected interactively in the SDR App on the cloud side using a WebSocket connection. In-processing data for the SDR App running on the cloud side is sent to the SDR App running on the client side via TCP communication. Because various communication functions for SDR Apps are added during the split processing of the flowgraph, the user need not be aware of this when editing it.

3.2.1 API for the UI Functions

GRC is composed of three packages: "base," "python," and "gui." We implemented the server package with the gui package replaced. The basic processing structures and data structures of other packages have not been changed when replacing the package. The main function of the server package is to provide the API. API functions have an almost one-to-one correspondence with the actions of GRC's UI action handler. For each object on the flowgraph, we implement a unique ID with a hash value of the memory address of the internal object of each element using SHA-1. This unique ID is used when manipulating each element in the flowgraph using the API.

3.2.2 Communication Functions for Data and Instructions

As mentioned in Section 3.1.2, the proposed implementation had three communication functions: in-processing data, parameter-update, and API communication.

in-processing data communication

It is used in the Task PDC of the SDR App. This communication function is used to transmit and receive signal sample data being processed between the distributed and executed SDR App. Use the existing TCP source/sink block of the GNU Radio. It is the same as Ref. [3] using TCP communication. These blocks are an implementation of a simple TCP socket by the asynchronous communication library of Boost C++ Libraries. The in-processed data are always transmitted only in one direction from the cloud side to the client PC side.

parameter-update communication

It is used in the Task PDC of the SDR App. Parameter-update communication is used for the parameter synchronous communication between the SDR Apps, which are divided and executed on both sides of the server/client when the parameter change is made from the widget of the SDR App under execution by user operation. This communication function was implemented using WebSocket as the communication protocol and JSON-RPC 2.0 [14] as the messaging protocol.

API communication

It is used for API communication between the server/client GRC. This communication function is used for API communication between the client and server GRCs. This communication function has the same implementation as the parameter-update communication function and implemented using WebSocket and JSON-RPC 2.0.

We describe WebScoket and JSON-RPC used for implementing parameter-update communication and API communication as follows:

- WebSocket is suitable for editing flowgraphs where frequent communication is needed because it does not incur HTTP overhead. WebSocket was also adopted in Refs. [10] and [13]. We used the Autobahn|Python WebSocket library [15]. This library is also used for implementations in Java (Android), C++ and JavaScript, and it provides a test suite. The Autobahn|Python WebSocket must be used with asynchronous I/O or an event-loop library. Hence, we used trollius, a backport library for Python 2.7.
- JSON-RPC is a simple and lightweight text-based protocol. This is a communication protocol that supports text-based protocols including WebSocket.

  Figure 3 shows an example of JSON request/response used in JSON-RPC 2.0. The members of JSON based on the specification of JSON-RPC 2.0 are described below:

  jsonrpc (request/response): This is a version of the JSON-RPC protocol.

  id (request/response): This is the request ID specified by the client. In our implementation, it consists of the API method name + @ + integer random number.

  method (request): This is an API method name.

  params (request): This is represented as an array or dictionary type specified for each API.

  result (response): This is required if processing succeeded. If the return value is necessary, it is held in dictionary type. In our implementation, if it is unnecessary, the character string is set to "Success."

  error (response): This is a dictionary-type member added instead of "result" only if it failed to process the request. It holds an error code in "code," an error message in "message," and error additional information in "data."

Source code 1: Request JSON object

```
{
  "jsonrpc" : "2.0",
  "id" : "method_name@12345",
  "method" : "api_name",
  "params" : {"param1": "value"}
}
```

Source code 2: Response JSON object

```
{
  "jsonrpc" : "2.0",
  "id" : "method_name@12345",
  "results" : "Success"
}
```

Source code 3: Error response JSON object

```
{
  "jsonrpc" : "2.0",
  "id" : "method_name@12345",
  "error": {
    "code": -32xxx,
    "message": "error",
    "data": "additonal information"
  }
}
```

Fig. 3 Examples of request/response JSON object.

We describe the initialization processing of the client-side program (Client GRC, SDR App on the Client PC side) at the time of connection in the API communication and parameter-update communication function. First, when the client side program establishes the connection of WebSocket, execute the `list_methods` method with JSON-RPC to obtain the method list of the API prepared by the server side program, then register it in the client-side program. Particularly on the parameter-update communication function used in the SDR App, in the SDR App on the cloud side, a parameter-update request must be associated from the SDR App in the Client PC side with any number of blocks and any number of parameters. Calling the registered API method with the parameter-update communication is based on the callback method, and it operates when the GUI widget (setter method) operates the same block name in the cloud-side SDR App. In the API communication function used in the Client GRC, execute `export_available_block_key_list` after `list_methods` to obtain a list of available blocks and register them in the client. After completion of the initialization processing, various operations can be executed on the Task PDC SDR App and Server/Client GRC using the registered API method.

3.2.3 Selecting Block Execution Location

In introducing the Task PDC of the SDR App, the new configuration value RunningMode is added to the block-definition file. RunningMode is introduced for the following purposes:

- identify the execution location of each block and identify the boundary when the user arbitrarily designates the split position of the flowgraph;
- identify the parameters to be synchronized by dependency among the flowgraphs constituting the SDR App to be distributed and executed (the actual synchronization function has already been explained in Section 3.2.2); and
- identify a static execution place by identifying a block with a restricted execution environment (e.g., a GUI block on which execution on the Client PC with the GUI environment is premised).

The configuration value of RunningMode is set by one of "Server," "Client," and "Variable."

The value "Server" is attached to a block to be executed on the cloud side. The value "Client" is added to the block to be executed on the client PC side. In the case of a sink block having a GUI function that is restricted in the execution environment, "Client" is set as a fixed value of RunningMode. The value "Variable" is statically specified in advance in the block

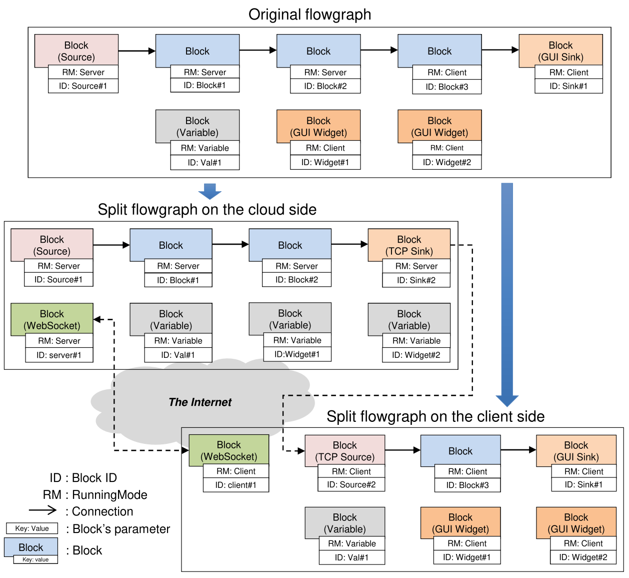


Fig. 4 Overview of Flowgraph Splitting.

that plays the role of the parameter and the variable. In the block where the characteristics of the GUI and variables coexist like `QT GUI Chooser` and `QT GUI Check Box`, the setting of "Variable" takes precedence over "Client". The field value of the block, in which the value of RunningMode is set, is always identified by the split generation function as the block having the field value to be synchronized in the distributed computing of the SDR App. For blocks that do not correspond to the static RunningMode configuration, the user can arbitrarily specify any value of "Server" or "Client." The split position is identified from the setting value of RunningMode of the block at both ends of the connection between the blocks. The split position is identified as the connection whose setting value of the source side block is "Server" and the setting value of the block of "sink" side is "Client" by the split generation function.

3.2.4 Split SDR App Generation using Flowgraph Split

A function for generating a split SDR App was implemented on the Server/Client GRC by the modifying the generator module in GRC's "python" package and the Cheetah template for flowgraph generation. **Figure 4** shows an overview of the process for flowgraph split and SDR App generation for the Server/Client GRC, which is described in simplified steps below.

(1) When a generation is requested, using the RunningMode values, two SDR Apps are generated: one on the client side with the blocks whose RunningMode setting is "Client," and one on the cloud side with the blocks whose RunningMode setting is "Server."

(2) The TCP source block is connected to the beginning of the client-side flowgraph. On the cloud-side flowgraph, the TCP sink block is connected to the end of the flowgraph.

(3) The blocks with a RunningMode setting of "Variable" are copied to both flowgraphs. Here, the GUI widget block whose RunningMode setting is "Variable" is converted into a variable block and copied.

(4) The parameter-update block is added to both flowgraphs. The green block shown in Fig. 4 is the block inserted as the parameter-update block.

4. Evaluation

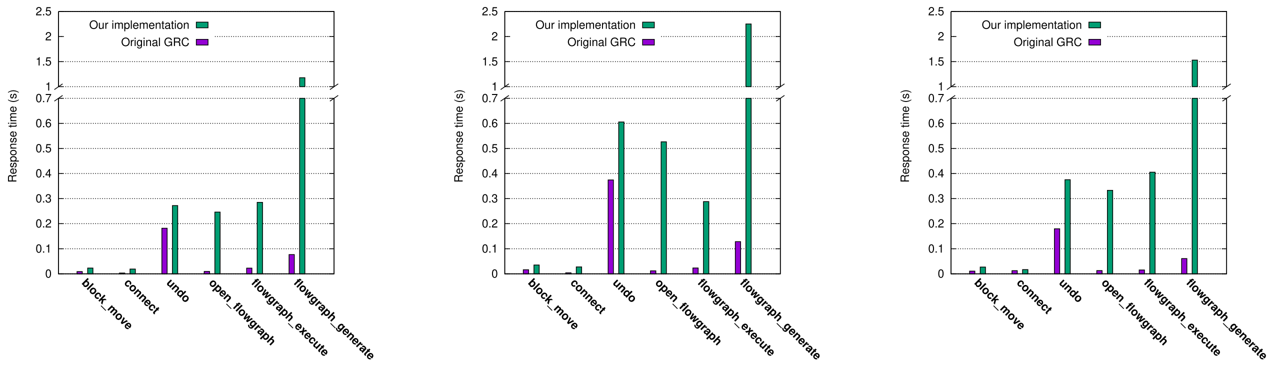
4.1 Compatibility of User Skills and Software Assets

To ensure the compatibility of software assets, we evaluated the compatibility of the proposed system with GRC and GR. We sought to determine whether changes were required to the module and file format and whether these changes were compatible. Specifically, we evaluated the compatibility of the blocks, the OOT modules, the .grc file, and the block-definition file. Existing blocks such as the output destination file or externally connected devices are restricted from executing to prevent unintended operations on the cloud side. When installing the GRC and the GR from the source code, with the exception of some of the essential libraries, only the block that corresponds to the available libraries on the system is built and installed. Specifically, 476 blocks were installed in the implementation environment, of which 65 blocks were restricted to prevent unintended operations in the cloud. In other words, 86.34% of the blocks maintained compatibility with GRC in both the Server and Client GRCs. In the OOT modules, the compatibility of a block is the same as the existing blocks. There is the same risk of the blocks being restricted in the cloud so that OOT modules will have unintended operations. It is difficult to confirm automatically whether the OOT module is compatible or which RunningMode value should be set for the OOT module's block when the user is free to deploy OOT modules they create to the client and the cloud. Therefore, the OOT modules do not have sufficient compatibility. The original block-definition file is not compatible with the Server/Client GRC because the block-definition file in the proposed implementation uses the additional RunningMode setting In regard to the .grc file, its compatibility is not affected because the original GRC ignores that the setting of RunningMode described in the .grc file. When the RunningMode setting is not described in the .grc file used by the Server/Client GRCs, both GRCs refer to the default RunningMode values in the block-definition file.

To evaluate the compatibility of the users' skill sets, we evaluated whether the implemented system can be operated using users' existing skills, and whether existing users require new knowledge for operating the proposed system. We did not change the UI of the Client GRC with respect to the UI of the original GRC. No significant changes were made to the functions assigned to each UI element. In addition, the user does not need to be aware of the communication functions between the client and the cloud regarding the split flowgraph and SDR App generation; almost no new knowledge is required for flowgraph editing except for RunningMode settings. The following knowledge about RunningMode settings is required: the RunningMode values for all blocks that constitute a loop must be kept together on either the server or client side. This is because processing data communications can occur in only one direction: from the cloud side to the client side. Although knowledge of the RunningMode settings is necessary, the compatibility of users' skill sets in the proposed implementation has been confirmed.

4.2 Response Time to User Operation

We measured the response time needed to complete processing



(a) mpsk_rx.grc (b) mpsk_stage6.grc (c) channelizer_demo.grc

Fig. 5 Response time for each user operation for three flowgraphs.

when a user operated the UI in the Original GRC and Client GRC. We then compared them with the response time required for client software. We measured six operations used to edit and execute flowgraphs in the GRC: "block_move," "connect" (connect the connectors of the two blocks), "undo," "open_flowgraph" (open a .grc file), "flowgraph_execute," "flowgraph_generate" (generate a .py file). These operations include minor interactions, file I/O, and take time to process the GRC internal flowgraph object. Measurements were made on three flowgraphs, which are the same as the flowgraphs used for the measurement In Section 4.3. The response time of the Original GRC and our implementation was calculated from the timestamp from when the handler of the GUI captured the user operation until the operation was returned to the user after the drawing update was complete. In our implementation, response time is the total processing time of Client and Server excluding network delay. We added source code to the Original/Server/Client GRCs to record timestamp in GUI/API event handler.

The reason for excluding the network delay is as follows: the network delay depends on the geographical arrangement of the Server and Client GRCs. In addition, the environment used for the measurement is a private cloud in the local area; both hosts are geographically very close; and the network latency becomes a measurement value, which is negligibly small. The general network latency is a minimum of 10 ms and a maximum of 300 ms depending on the geographical arrangement. In particular, the network latency at domestic and neighboring countries is 10 ms to 50 ms [16], [17]. Depending on the size of the sum of the processing time on both hosts against this network latency, whether the network latency or processing time becomes dominant in response time is different. In the evaluation of this measurement result, both hosts were assumed to be within domestic or neighboring countries. The abovementioned 10 ms to 50 ms was also used as a reference value of the network latency and evaluated.

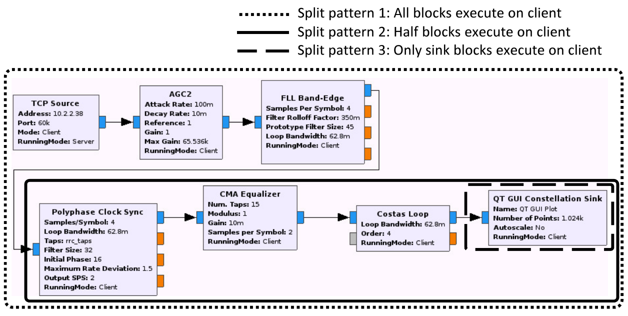
Studies in human–computer interaction have found that when the response time is below 150 ms, the productivity of the user is not affected [18]. For delays of 150 ms to 1 s, users gradually become aware of the delay. Delays above 1 s cause users to feel frustrated, which affects their productivity. Hence, we regard 1 s as the maximum acceptable response time for the client software.

Figure 5 shows the average measured response time of five trials. For "block_move," "connect," and "undo," our implementation takes nearly double the response time compared with the Original GRC. Because the Client GRC and Server GRCs process synchronously, this is a predictable result. In contrast, for "open_flowgraph," "flowgraph_execute," and "flowgraph_generate" take much longer to complete (almost double) than in the Original GRC. It took time to complete the "open_flowgraph" operation in the Client GRC, because it takes time to generate unique IDs on the Server GRC side and assign it to each element on the Client GRC side. With regard to the "flowgraph_execute" operation, it is necessary to listen to the socket of the SDR App on the cloud side; hence, the client side must wait for the startup of the SDR App on the cloud side to complete. The "flowgraph_generate" operation had a particularly slow response time. This is because it takes time for the flowgraph to be divided, as shown in Section 3.2.4, including verification of the RunningMode values of each block and operation such as addition and deletion of blocks based on the verified values. All user operations except for "flowgraph_execute" are linear time operations on the internal flowgraph object, and their response time is roughly proportional to the flowgraph size. The size of a flowgraph is determined by the number of blocks, connections, and parameters. When the operation response times are evaluated, the operations for the Original GRC are all within 1 s. In contrast, in our implementation, the operation of "flowgraph_generate" greatly exceeds 1 s, which could affect the user.

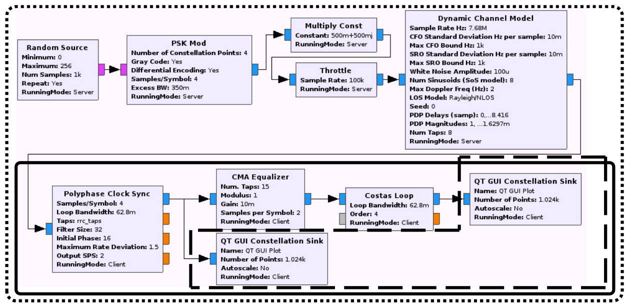
For lightweight operations that change one or two flowgraph elements such as "block_move" and "connect," which are frequently used, we found the response times even including network latency to be around 100 ms. This value is acceptable considering the response time as the client software. In contrast, except for lightweight flowgraph editing operations, the latency of the application dominates that of the network. Especially when the size of the flowgraph becomes large, the user's waiting time becomes long. Because GRC has no visual feedback until the processing is complete, some visual feedback to the user would be required if the response time of 1 s or more is expected.

4.3 Distributed SDR Processing

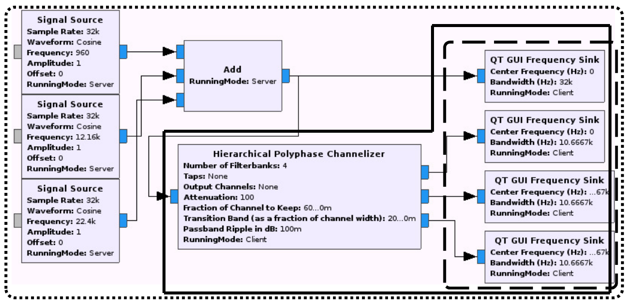
We conducted another set of experiments to investigate the effect of SDR App task PDC, realized by flowgraph splitting.


Split pattern 1: All blocks execute on client
Split pattern 2: Half blocks execute on client
Split pattern 3: Only sink blocks execute on client
TCP Source
AGC2
FLL Band-Edge
Polyphase Clock Sync
CMA Equalizer
Costas Loop
QT GUI Constellation Sink

(a) mpsk_rx.grc (receiver)[19]



(b) mpsk_stage6.grc (simulation)[20]



(c) channelizer_demo.grc (simulation)[21]

Fig. 6 Three flowgraphs used for evaluation and their split patterns.

We investigated the computational resource utilization of several flowgraphs with different split patterns, adjust the computational resource consumption for a single computer, and confirm that computational resource constraints can be avoided. It is possible to provide an SDR App execution environment that does not depend on the computational resources available to a user local by adjusting the computing resources used by a single computer.

The scenario of the investigation assumes that the user creates and executes the SDR App including some plotting (e.g., a waveform, spectrogram, or constellation diagram) as output, which means that the termination of the data flow in the flowgraph is always on the user's client PC. Three flowgraphs (mpsk_rx.grc [19], mpsk_stage6.grc [20], and channelizer_demo.grc [21]) were selected from GR examples assuming use for simulation, prototyping, sampling data analysis. We measured performance and computational resource utilization for the three split patterns with varying ratios of client to cloud blocks. **Figure 6** shows three selected flowgraphs and three division patterns displayed on the GRC. (a) is the signal reception process of decoding a signal and rendering a constellation. (b) is the transmission/reception simulation, including the processing of a propagation model from signal generation. (c) is the simulation of a channelizer that divides the input signals into single sub-band signals through the polyphase filter bank. Particularly in (b), rich calculation resources are required, especially for complicated serial processing for signal generation and propagation model calculation. (c) has a large number of sample data; hence, the memory bandwidth becomes a bottleneck on the shared memory system, and communication between the cloud and the client PC becomes a bottleneck at the time of distributed processing.

Performance evaluation criteria are throughput, which is the Mega number of samples processed per second (M samples/s), and latency, which is the time interval it takes for a sample to be processed from a source block to a sink block. Throughput was measured on the flowgraph by adding a measurement block to the end located farthest from the source block, and the latency as a time interval was calculated from the time stamp given to the sample in the block added just after source to the time stamp at the same position as the throughput measuring block. Latency is the sum of latency in each of the SDR Apps divided into the client and the cloud. Note that the latency does not include the latency of TCP communication between the client and the cloud. The client PC was equipped with an Intel Core i5-2400 4-core (four threads) 3.10 GHz CPU and 8 GB RAM running Debian 9.1. The cloud was equipped with an Intel Xeon E5-2643v4 3.40 GHz 6-core (12 threads) CPU and 126 GB RAM running Ubuntu 14.04 LTS. Virtualization was not used. These machines were connected by the laboratory's Gigabit Ethernet network. We used the "pidstat" Linux command to acquire %user (the CPU utilization of the user program (SDR App)) and %system (the CPU utilization of the kernel) for each thread. In addition, we used the "top" command to obtain the memory usage of the SDR App and the "tiptop" command to obtain the instructions per cycle (IPC).

Table 1 shows the measurement results for each split pattern of each flowgraph. The measurement results for %user, %system, RSS, VSZ, and IPC are the average values for attempts to acquire data for 10 trials during SDR App execution. The CPU utilization (%user) is allocated between the cloud and the client as intended in each split pattern. Regarding RSS and VSZ, significant results were not obtained because most memory was occupied by the GUI library. In addition, because the JSON-RPC client and server used for parameter-update communication in the SDR App used about 65 MiB RSS and 850 MiB VSZ respectively, the actual usage increases. The buffer size between blocks in the GNU Radio is equal to the OS page size (32 KB or 64 KB); thus, a large difference in memory usage does not follow. However, we believe that the shared memory state is eliminated by the SDR App execution host distribution, and we think that the memory bandwidth constraint is partially improved. We were successfully able to adjust the computational resource (CPU) utilization on a single computer.

We describe the performance at runtime of task PDC according to the flowgraph split. Improvement in throughput can hardly be expected. Table 1 (a) shows a slight improvement in throughput when splitting the flowgraph, whereas Table 1 (c) shows that the throughput is substantially lowered because of network bandwidth restriction. Latency is suppressed after splitting the flowgraph. Table 1 (a) shows an improvement of up to about 0.00265 s, Table 1 (b) has deferent scale, but does not show im-

Table 1 CPU utilization, throughput, and latency of each SDR App for each split pattern.

(a) mpsk_rx.grc

Split pattern	Execute on	%user	%system	RSS (KiB)	VSZ (KiB)	Throughput (M samples/s)	Latency (s)	IPC
1	Client PC	188.5	4.8	142,208	1,314,016	1.20288	0.004015	1.25
2	Client PC	100.3	9.9	141,888	1,166,028	1.29864	0.001726	1.21
	Cloud	116.3	13.6	93,564	1,169,552			1.42
3	Client PC	5.8	8.6	140,964	944,920	1.27740	0.001367	0.65
	Cloud	236.8	19.2	94,512	1,391,292			1.43

(b) mpsk_stage6.grc

Split pattern	Execute on	%user	%system	RSS (KiB)	VSZ (KiB)	Throughput (M samples/s)	Latency (s)	IPC
1	Client PC	160.3	51.9	146,116	2,133,300	0.02516	18.8496	0.88
2	Client PC	4.9	0.7	142,784	1,240,392	0.02489	18.8511	1.10
	Cloud	141.7	69.4	99,460	1,915,032			0.90
3	Client PC	1.5	0.6	142,260	1,093,152	0.02497	18.8499	1.24
	Cloud	145.0	70.9	101,836	2,210,616			0.95

(c) channelizer_demo.grc

Split pattern	Execute on	%user	%system	RSS (KiB)	VSZ (KiB)	Throughput (M samples/s)	Latency (s)	IPC
1	Client PC	294.4	32.6	141,788	1,968,344	11.27090	0.002151	2.03
2	Client PC	87.2	26.4	151,196	1,839,268	2.31227	0.001379	1.45
	Cloud	31.8	9.9	93,348	1,317,080			2.42
3	Client PC	17.5	27.8	150,456	1,395,328	2.00368	0.001878	0.63
	Cloud	159.4	27.0	100,764	2,056,628			1.71

provement, and Table 1 (c) shows an improvement of up to about 0.00077 s. Table 1 (b), which does not show much improvement in latency compared with the unsplit SDR App, the value of IPC at runtime is 0.88 that is less than 1, and the value of %system at runtime is large. This also has no change in split patterns 2 and 3, where the main processing was moved to the cloud. Hence, in the case of %system is high and IPC is less than 1, the SDR App includes a process of stalling, which is considered to be a poor improvement in latency. Performance improvement on latency can be expected in thread-level task PDC, but it depends on the implementation of the SDR App itself. Moreover, a SDR App that can expect the effect of task PDC can be determined from the trend of resource utilization such as CPU utilization as well as IPC during the run time. In the case where each step of the serialized wireless signal processing is executed in parallel by a plurality of CPU cores of the cloud, like a (c), as a step becomes a bottleneck, the whole process is rated-limiting. As shown in (b), the network bandwidth becomes a throughput constraint. Therefore, in the SDR Task PDC using the cloud, an application composed of blocks (implementation of steps) that can provide processing performance according to the CPU performance is suitable, and it is good when the split position is not subject to the restriction by the network bandwidth.

5. Conclusion

In this paper, we proposed an implementation of a remote SDRDE based on the GRC to realize a cloud-based platformization of an SDR environment. The proposed system provides an SDR environment to an unspecified number of users because the UI is separated from the GRC, and it enables the constraints of computational resources to be avoided by transferring the SDR App execution platform to the cloud and utilizing task PDC.

Our evaluation results showed that the remote SDR execution proposed by the GRCWG could be implemented. With this implementation, it was challenging to ensure software compatibility and flexibility for diverse computer configurations. With this implementation, maintaining compatibility of software asset and user skill sets, and avoiding computational resources constraints on a single host, was our challenge. Although there are restrictions designed to prevent unintentional operations on some blocks, compatibility of software asset was generally maintained. We confirmed that the proposed implementation is compatible with the current user skill set. Using existing software assets, to avoid the limits of computational resources available to the user on a single computer, we developed task PDC to allocate resources between the client and cloud at the user's discretion. Task PDC has the potential to improve latency, and from the relationship between CPU utilization and IPC values at runtime, the trend of the performance of the SDR App which this is generally effective was found.

One of the limitations of this study is that the user is restricted from freely introducing OOT modules. An enhanced OOT management function tailored to the platform is required. Although, in this paper, we focused on the development and execution environment of SDR within the whole platform, by realizing our other platform function, the proposed SDR system will contribute to open innovation.

Acknowledgments This work was supported by JSPS KAKENHI Grant Numbers 16K16042 and the MIC/SCOPE #155003007.

References

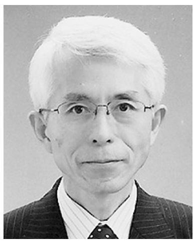
[1] Uehara, K.: Research and Development of Software Defined Radio and Cognitive Radio Technologies, *IEICE Trans. Communications*, Vol.J100-B, No.9, pp.693–704 (2017).

[2] Sugano, H., Miyamoto, R. and Okada, M.: Fully Software-Based Real-Time Digital Terrestrial Boradcasting Receiver Using a GPU, *IEICE Trans. Information and Systems*, Vol.J95-D, No.5, pp.1216–1224 (2012).

[3] Tracy II, T. and Stanand, M.: GR-Router: A Distributed GNU Radio

Framework (online), available from ⟨https://github.com/tjt7a/GR-Router/blob/experimental/docs/GR_Con_Presentation_tracy.pdf⟩ (accessed 2017-11-13).
[4] Checko, A., Christiansen, H.L., Yan, Y., Scolari, L., Kardaras, G., Berger, M.S. and Dittmann, L.: Cloud RAN for Mobile Networks —A Technology Overview, *IEEE Communications Surveys Tutorials*, Vol.17, No.1, pp.405–426 (2015).
[5] Suzuki, H., Kawakita, Y. and Ichikawa, H.: Remote Implementation of GNU Radio-based SDR Development Environment, *Proc. 22nd Asia-Pacific Conference on Communications* (*APCC 2016*), pp.355–360 (online), DOI: 10.1109/APCC.2016.7581497 (2016).
[6] Miller, R., Xu, W., Kamat, P. and Trappe, W.: Service Discovery and Device Identification in Cognitive Radio Networks, *4th Annual IEEE Communications Society Conference on Sensor*, *Mesh and Ad Hoc Communications and Networks*, *SECON '07*, pp.670–677 (2007).
[7] Mizutani, Y., Sato, M., Kawakita, Y. and Ichikawa, H.: Dynamic Spectrum Sensing for Energy Harvesting Wireless Sensor, *2013 IEEE 11th International Conference on Dependable, Autonomic and Secure Computing* (*DASC*), pp.427–432 (2013).
[8] Katti, S., Gollakota, S. and Katabi, D.: Embracing Wireless Interference: Analog Network Coding, *SIGCOMM Computer Communication Review*, Vol.37, No.4, pp.397–408 (2007).
[9] Gollakota, S. and Katabi, D.: Zigzag Decoding: Combating Hidden Terminals in Wireless Networks, *Proc. ACM SIGCOMM 2008 Conference on Data Communication*, *SIGCOMM '08*, pp.159–170, ACM (2008).
[10] Sharakhov, N., Marojevic, V., Romano, F., Polys, N. and Dietrich, C.: Visualizing Real-time Radio Spectrum Access with CORNET3D, *Proc. 19th International ACM Conference on 3D Web Technologies*, *Web3D '14*, pp.109–116, ACM (2014).
[11] Newman, T., Shajedul Hasan, S., DePoy, D., Bose, T. and Reed, J.: Designing and deploying a building-wide cognitive radio network testbed, *IEEE Communications Magazine*, Vol.48, No.9, pp.106–112 (2010).
[12] GNU Radio: GRCroadmap - gnuradio.org (online), available from ⟨https://wiki.gnuradio.org/index.php/GRCroadmap⟩ (accessed 2017-11-06).
[13] Google: Google Summer of Code Archive - gr-bokehgui: A Web based GUI for GNU Radio applications (online), available from ⟨https://summerofcode.withgoogle.com/archive/2017/projects/5129007263645696/⟩ (accessed 2017-11-06).
[14] JSON-RPC Working Group: JSON-RPC 2.0 Specification, available from ⟨http://www.jsonrpc.org/specification⟩ (accessed 2018-09-18).
[15] Crossbar.io: Autobahn|Python — AutobahnPython 0.10.7 documentation (online), available from ⟨http://autobahn.ws/python/⟩ (accessed 2017-11-13).
[16] AT&T: Global Network Latency Averages (online), available from ⟨http://ipnetwork.bgtmo.ip.att.net/pws/global_network_avgs.html⟩ (accessed 2018-09-18).
[17] Verizon: IP Latency Statistics (online), available from ⟨http://www.verizonenterprise.com/about/network/latency/⟩ (accessed 2018-09-18).
[18] Tolia, N., Andersen, D.G. and Satyanarayanan, M.: Quantifying interactive user experience on thin clients, *Computer*, Vol.39, No.3, pp.46–52 (online), DOI: 10.1109/MC.2006.101 (2006).
[19] Rondeau, T.: An Approach to Digital Demodulation (online), available from ⟨https://static.squarespace.com/static/543ae9afe4b0c3b808d72acd/543aee1fe4b09162d08633d9/543aee20e4b09162d0863523/1409511797677/rondeau-03-digital_demodulation.pdf⟩ (accessed 2017-11-13).
[20] Rondeau, T.: GNU Radio: digital demodulation example scripts (online), available from ⟨https://static.squarespace.com/static/543ae9afe4b0c3b808d72acd/543aee1fe4b09162d08633d9/543aee20e4b09162d0863524/1409511836183/03-digital_demodulation.tar.gz⟩ (accessed 2017-11-13).
[21] GNU Radio: Github.com - gnuradio/channelizer_demo.grc at master ·gnuradio/gnuradio (online), available from ⟨https://github.com/gnuradio/gnuradio/blob/master/gr-filter/examples/channelizer_demo.grc⟩ (accessed 2017-11-01).

Hirotaka Suzuki received his B.Eng. degree from the University of Electro-Communications in 2016. He is a Master's student at the University of Electro-Communications. His current research interests are software-defined radio and information technology.



Haruhisa Ichikawa received his B.S., M.S., and Dr. Eng. degrees in electrical engineering from the University of Tokyo in 1974, 1976, and 1989, respectively. He joined NTT Laboratories in 1976, where he was engaged in fundamental research on communications software and distributed computing. He created and conducted many R&D projects for software, Internet, information sharing platform, and ubiquitous networks, including business incubation. He was Executive Director of NTT Science and Core Technology Laboratory Group till 2007, and is currently Professor Emeritus at the University of Electro-Communications, Tokyo.

Jin Mitsugi received his B.S. degree from Nagoya University in 1985, and his M.S. and Ph.D. degrees from Tokyo University in 1987 and 1996, respectively. He was with NTT Laboratory from 1987 pursuing research and development on satellite communication systems. He has been with the Auto-ID Laboratory, at Keio University, Japan, since 2004. His research interests are network RFID, sensor network systems, satellite communications, and operations research.

Yuusuke Kawakita received his B.A., M.A. degrees and Ph.D. from Keio University in 2000, 2002 and 2008, respectively. He is an associate professor at Kanagawa Institute of Technology. His present research interests focus on the ubiquitous sensing and its platform architecture.