2020 IEEE 17th International Conference on Mobile Ad Hoc and Sensor Systems (MASS) | 978-1-7281-9866-8/20/$31.00 ©2020 IEEE | DOI: 10.1109/MASS50613.2020.00073

TinyCSI: A Rapid Development Framework for CSI-based Sensing Applications

Yuxiang Lin, Wei Dong, Bingji Li, Yi Gao
College of Computer Science, Zhejiang University, China
Alibaba-Zhejiang University Joint Institute of Frontier Technologies
Email: *linyx@zju.edu.cn, dongw.cs@gmail.com, lbj123kst@zju.edu.cn, qhgaoyi@gmail.com*

***Abstract*—Channel State Information (CSI)-based wireless sensing has recently attracted extensive attention from both academia and industry. However, it is still challenging and time-consuming to develop a CSI-based sensing application due to the use of complex signal processing algorithms and the requirements of accuracy and responsiveness. In this paper, we present TinyCSI, a rapid development framework for CSI-based sensing applications. With TinyCSI, developers only need to write a main script to determine the CSI collection settings and a callback function to process the collected CSI signals using the well-abstracted Matlab/C-based library, without dealing with the connection/transmission details of the sensing nodes. To achieve fast performance tuning, TinyCSI also provides three working modes for different deployment requirements: a remote mode for fast iteration of the sensing algorithms and their parameters, an efficient mode for making full use of computing resources and improving sensing responsiveness, and a standalone mode for offline running sensing systems on individual nodes. We implement three representative demos and conduct real-world user studies to show the workflows and benefits of TinyCSI. Experimental results show that TinyCSI helps reduce the lines of code significantly compared to the original implementation. More importantly, the efficient mode can generate an optimal computing resource allocation solution and significantly improve the sensing responsiveness.**

I. Introduction

When the transmission of wireless signals is affected by adjacent objects, analyzing the wireless signal can provide sensing capability for the objects. This radar-like technique is known as wireless sensing, and has been under active research during the past ten years. In particular, researchers found that the fine-grained Channel State Information (CSI) provided by Commercial Off-The-Shelf (COTS) Wi-Fi devices [1, 2] could be used in many sensing applications, including human detection [3–7], activity recognition [8, 9], gesture recognition [10, 11], localization and tracking [12–16].

Although many sensing algorithms have been proposed for various applications, developers are still confronted with two difficulties in practice:

First, it is *time-consuming* to *rapidly* build novel sensing systems or customize existing ones for specific scenarios. Existing CSI-based sensing systems are tightly coupled with their applications. This coupling requires developers to know all stages for CSI signal processing including 1) preprocessing, 2) motion detection and segmentation, and 3) other application-specific algorithms. Among these stages, each of them has plenty of different algorithms and details.

Second, it is *difficult* to *fast* tune various system options and execution modes to achieve high sensing accuracy and high sensing responsiveness. It is labor-intensive to try every possible approach even for an experienced developer with detailed knowledge of CSI processing algorithms. For a junior developer, it is even harder for her/him to understand how one stage in the implementation affects the final performance.

To address the above two difficulties, we propose TinyCSI, a framework for the rapid development of CSI-based sensing applications.

To *enable rapid development*, we clearly split sensing data collection and data processing in TinyCSI: users are only required to define a main script for determining CSI collection settings of the node and build a callback function for essential CSI signal processing. More importantly, we have implemented a sensing library covering 11 different API categories, after investigating more than 30 existing CSI-based sensing applications. Our library covers the three stages of CSI signal processing and is rich enough to support more than 60% apps from the literature. Developers can quickly invoke these APIs for rapid implementation. We also allow extending our library with customized APIs or by incorporating other third-party API implementations.

To *facilitate performance tuning*, TinyCSI provides different working modes: In the remote mode where all signal processing is performed at the server-side, developers can focuses on trying different algorithms or tuning different parameters to achieve the best sensing accuracy. In the efficient mode, TinyCSI can automatically partition different components (i.e., APIs) between the sensing device and the server, resulting in the highest sensing responsiveness while preserving the sensing accuracy. Compared with previous CSI-based sensing work, this mode is a major advantage of TinyCSI. An additional standalone mode is also provided for cases where server processing is not available.

We have implemented three representative demos using TinyCSI as references for developers. These demos cover the three major categories of CSI-based sensing applications, i.e., motion detection, location awareness and CSI fingerprinting. These demos show the general system development process and some implementation details using TinyCSI. The implementations of the three demos show that TinyCSI helps reduce up to 93.9% of the lines of code on average. We evaluate the performance of the efficient mode since it is

2155-6814/20/$31.00 ©2020 IEEE
DOI 10.1109/MASS50613.2020.00073

TABLE I: **Three major types of CSI-based sensing applications.**

Motion detection			Location awareness			CSI fingerprinting		
System	Method	Scenario	System	Method	Scenario	System	Method	Scenario
FIMD [7]	Cluster	Lab & corridor	Pilot [15]	Maximize a priori probability	Lab & lobby	CARM [8]	Hidden Markov Model	Lab & apartment
SIED [4]	Hidden Markov Model	Office & meeting room	D-MUSIC [12]	Revised MUSIC algorithm	Meeting room, office & lobby	BodyScan [9]	Support Vector Machine	Lab & outdoor
PADS [6]	Support Vector Machine	Classroom, office & corridor	LiFS [16]	Solve a power fading model	Home, library & classroom	WiAG [17]	k-Nearest Neighbors	Lab
Omni-PHD [3]	Earth Mover's Distance	Hall & lab	FILA [14]	Solve a distance model	Chamber, lab, hall & corridor	WiFinger [10]	Dynamic Time Warping + k-Nearest Neighbors	Lab
DeMan [5]	Threshold-based	Classroom & lab	SpotFi [13]	MUSIC algorithm	Whole floor of a building	SignFi [11]	Convolutional Neural Network	Lab & home

a major advantage of TinyCSI. The remote and standalone modes *without fast tuning* are usually the development modes of previous work. Our experimental results show that for the three representative demos, the efficient mode decreases the sensing delay by 33%-44% relative to the standalone mode, and 9%-18% relative to the remote mode. Real-world user studies have shown that TinyCSI significantly reduces their development time for building their prototype systems. Results also show that TinyCSI incurs acceptable overhead in terms of execution time and program space.

We summarize the contributions of this paper as follows.

(1) We present TinyCSI, the first rapid development framework for CSI-based sensing applications. We have made TinyCSI open source [1].

(2) We provide an efficient mode and formulate the resource allocation process as an optimization problem to optimize the sensing responsiveness. We also provide two other modes, e.g., a remote mode and a standalone mode, to facilitate performance improvement in different development phases.

(3) We implement three representative CSI-based sensing applications and conduct user studies to evaluate TinyCSI. Results show that TinyCSI can significantly reduce the lines of code and sensing delay, without introducing much overhead.

II. Related Work

A. CSI-based Sensing Application

In modern Wi-Fi networks, the communication channel using Orthogonal Frequency Division Multiplexing (OFDM) comprises of multiple orthogonal subcarriers at different frequencies [18]. CSI captures how wireless signals propagate from the transmitter to the receiver at a granularity of the subcarrier level. CSI-based sensing systems record the CSI of one or more wireless links and then provide context-aware computations. As shown in Table I, we classify existing CSI-based sensing applications into three major categories, i.e. motion detection, location awareness and CSI fingerprinting.

Motion detection systems continuously detect whether there is an object walking in the monitoring area based on various indicators and algorithms [3, 5]. Motion detection usually serves as the basic module in complex CSI-based sensing systems for saving energy.

Location awareness systems can either localize an object in the monitoring area or further track the continuous movements of the object. Many recent works extract Angle-of-Arrival (AoA) from CSI measurements to localize a target [13] while some work directly extracts the position information with different mathematical models (e.g., power fading model [16] and distance model [14]).

CSI fingerprinting systems map the extracted features to different activities [8] or gestures [11] to realize novel sensing applications such as activity identification and gesture recognition. The property is that different actions will lead to different frequency-selective wireless fading, which can be well characterized by the fine-grained CSI.

The algorithm selection and parameter determination process of the above sensing systems usually take a significant amount of effort. Moreover, most of them are tested in a limited number of scenarios. Once the environment has changed, system parameters or even some algorithms need to be updated to accommodate the new scenario. The lack of rapid development hampers the deployment of these sensing systems. TinyCSI is designed to solve these problems by providing appropriate API abstractions and a user-friendly GUI for fast tuning of algorithms and parameters.

B. Rapid Development System

There exist several rapid development libraries/frameworks/platforms. However, all of them have very different goals from TinyCSI. TinyLink [19] is a holistic system for the rapid development of IoT applications and uses a top-down approach for both hardware and software designs. CITA [20] and CAreDroid [21] are rapid development systems for applications on smartphones. LibAS [22] is a cross-platform framework to ease the development of acoustic sensing apps. Different from them, TinyCSI aims to facilitate the development of CSI-based sensing systems. In addition, the sensing algorithms of these systems entirely run either on the sensing devices or on the server. Differently, we have designed an efficient working mode in TinyCSI to help make full use of computing resources and reduce the overall sensing delay.

To the best of our knowledge, TinyCSI is the first framework where developers can rapidly build their CSI-based sensing applications. An important advantage of TinyCSI is that it provides three different working modes for developers. After the rapid validation of their sensing algorithms, developers can rapidly deploy their systems in one of the three working mode to meet different deployment requirements.

III. TinyCSI Overview

Figure 1 illustrates the overview of TinyCSI. Developers only need to develop a main script to determine the CSI collection settings for sensing node and a callback function to process the continuous CSI signals.

[1]TinyCSI: https://github.com/ZELK001/Tinycsi-csirapiddevelopment

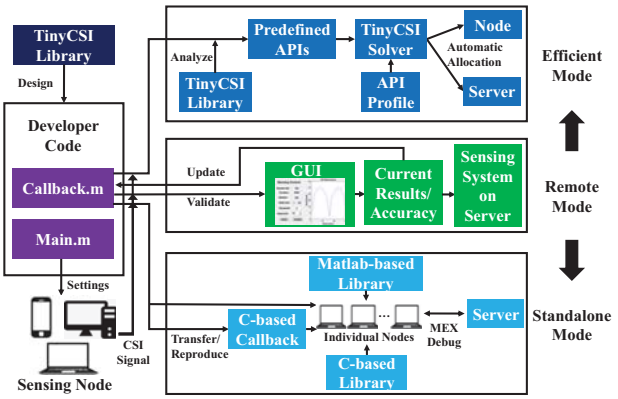

Fig. 1: **TinyCSI overview.**

The developers can rapidly validate their algorithms and parameters in the remote mode, and then optionally finalize the system to the efficient mode or the standalone mode according to the requirements.

In the remote mode, TinyCSI provides a rapid prototyping environment, including a GUI for visualization and an underlying SensingServer structure for control. The remote mode can help developers intuitively make aware of underlying channel changes and their dependence on various factors (user movement, environment changes, etc.). With the API library and visualization of TinyCSI, developers can improve their fundamental understanding of CSI sensing and rapidly determine their callback functions (including both algorithms and parameters) to achieve good sensing accuracy.

The efficient mode is a major advantage of TinyCSI. In the efficient mode, our framework is able to optimize the resource allocation scheme for the designed application. Specifically, TinyCSI first links the callback function with predefined APIs in the library and then provide a TinyCSI solver for resource allocation. TinyCSI solver puts all detected APIs and their predefined profiles (including the time, memory and energy cost of each API on each computing resource) as its input, and automatically allocates these APIs to their appropriate places for processing.

In the standalone mode, the callback function will be entirely transmitted to individual nodes and can be directly executed with the Matlab library of TinyCSI. For devices that only support C, TinyCSI can automatically transfer the developed Matlab callback function to C. Developers can also use the C-based TinyCSI library to reproduce the callback function. TinyCSI integrates the MEX tool [23] in Matlab to help remotely tune the C-based callback function.

IV. TinyCSI Design

In this section, we will first introduce the library design in TinyCSI. Then we will describe the three working modes of TinyCSI in detail and its expected development flow.

A. Extensive library

To provide widely-used APIs, we have investigated more than 30 existing CSI-based sensing systems that belong to the three major application types listed in Table I. We observe that these systems usually need a preprocessing stage and a motion detection stage before the essential sensing stage (if any). We began with a coarse-grained decomposition for each stage of CSI signal processing. However, a black box design like *CSISensing.onMotionDetection(threshold, callback)* will lose the flexibility of adding extra signal processing algorithms and enable little code reuse. We then progressively split these components up in order to reach the desired granularity for further reuse. For example, the activity recognition system CARM [8] and the user identification system WiID [24] both use a PCA denoising approach, which can therefore be reused. TinyCSI allows developers to preserve the programmability for customizing sensing algorithms via providing 42 basic and commonly used signal processing APIs in the library in both Matlab and C. As shown in Table II, we group them into 11 categories by the functionalities. Our current APIs can cover more than 60% of the investigated systems. Each category contains interchangeable components for developers to choose. For example, while CSI-based sensing applications usually require a lot of calibration to reduce environmental noise and interferences, TinyCSI provides seven kinds of filters (e.g., Butterworth filter, PCA filter, etc.) in the Filter category for signal preprocessing. Developers can select the most suitable filter for their applications after multiple trials. The remaining uncovered systems include either infrequently invoked APIs or complex algorithms that we have not implemented yet due to the limited development time.

The library can be easily extended since all the APIs are based on standard C and Matlab. For developers who want to add their own components, they only need to encapsulate their self-designed algorithms into function files (i.e., .c file or .m file) with input and output in the required format. We believe that our library can have important implications and guidance for developing new APIs. Developers can also make improvements based on our APIs since we have made the entire library open source. For example, if a developer want to extend the MUSIC algorithm to provide both AoA and ToA estimations [12, 13], she/he only needs to rewrite the steering vector, the received signal vector, and the spectrum function in the original *MUSIC* API, which is much faster than writing from scratch. In addition, many third-party libraries, e.g., LibSVM [25], can be directly integrated into TinyCSI due to the compatibility of C and Matlab. After being integrated into TinyCSI, these extended library functions can also benefit from the three working modes for fast performance tuning.

B. Developer Code

The *main* script should include the basic collection parameters, such as sample rate and duration. Necessary parameters for the callback function, such as motion detection threshold, can also be initialized in the main script. Among these settings, parameters that will affect the sensing results can be initialized as global parameters, which can later be updated with a GUI for fast tuning. At the end of the main script, a SensingServer object should be created with a predefined script *SensingServer.m* to invoke the callback function.

TABLE II: **API categories in TinyCSI.**

Sampling	Measurement	Filter	Phase	Subcarrier	Statistical Feature	Frequency Feature	Time Feature	Space Feature	Supervised Learning	Unsupervised Learning
1	6	7	2	2	10	3	2	2	5	2

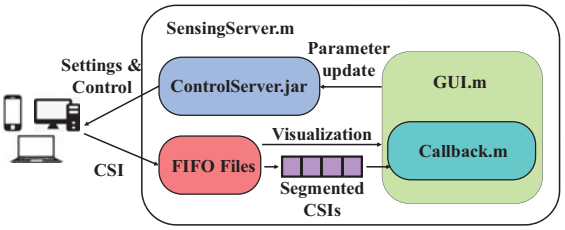

Fig. 2: **The structure of SensingServer.m.**

The *callback* function focuses on processing the collected CSI measurements and returning the sensing results. Developers can specify callback function logic using the well-abstracted API library, ignoring the node connection/transmission details. They only need to develop some glue code to combine the required APIs for their systems (see Section VI-A for how to design a main script and a callback function).

C. Remote Mode

In the remote mode, TinyCSI will first create a SensingServer object with *SensingServer.m* to take over the entire sensing system based on the main script. The SensingServer object plays an important role in hiding the connection/-transmission details of the sensing nodes. Figure 2 shows the structure of the *SensingServer.m*. This SensingServer is a main primitive predefined in TinyCSI to trigger the callback function when CSI data is transmitted to the server. Inside the SensingServer, TinyCSI creates a Java socket interface exported as a *ControlServer.jar* file to control the sensing node. Through the socket interface, the server can send control commands (e.g., *start*, *update* and *stop*) and collection settings defined/updated in the main script to the node in the form of an automatically generated shell script. Then the node continuously collects and transmits the CSI measurements to the server. SensingServer will record the CSI measurements into a FIFO file and show them in the GUI. Besides, the GUI includes figures showing CSI changes and predefined features (if any) used in the callback. For a user-specified subcarrier, we will further show its detailed amplitude changes, frequency domain representation, and some basic signal information (e.g. SNR and variance).

Besides signal visualization, the GUI also supports the real-time update of some important parameters (i.e., CSI collection settings and global parameters) for a better sensing performance. When modifying the parameters (e.g. sample rate, time duration, etc.) in the text fields and click *Update* button in the GUI, *GUI.m* will trigger a built-in system call to send the new CSI collection setting to the node through the Java socket interface. Inside the *GUI.m*, TinyCSI uses the assigned callback function *Callback.m* to process the received signals. The received CSI data will be segmented into small pieces in the backend. The window size of each piece will affect both the sensing results and computation overhead, and should also be carefully tuned with the GUI. Finally, each segment will be sent to the callback function, which is responsible for calculating the sensing results. A conceptual callback function of a motion detection system can be: *flag* = *compare_threshold(get_var(filter(received_signal)))*. The detection flag (i.e. *true* or *false*) will be calculated for each signal segment and updated in the GUI. A complete code example of using TinyCSI to implement a real motion detection system is provided in Section VI-A.

Our remote mode design aims to help developers focus on building the essential sensing algorithm and tuning the parameters rapidly. We choose to implement the remote mode with Matlab because it provides several useful signal processing and visualization tools.

D. Efficient Mode

The efficient mode is designed to achieve the highest sensing responsiveness without losing sensing accuracy. In the efficient mode, TinyCSI will utilize the innate computational capability of both the sensing node and the server.

Determining where these separated APIs should be executed is essentially an optimization problem. This optimization is often ignored by previous CSI-based sensing work, which collect CSI data and transmit them online or offline to the server for later computation. Instead, TinyCSI succeeds in achieving the minimal sensing delay while preserving the sensing accuracy. Different from existing resource allocation approaches [26, 27], there are multiple special dependencies need to be considered in TinyCSI: 1) the APIs in a sensing task have dependencies, 2) the final results may depend on the outputs of multiple tasks (e.g. demo 2 in Section VI-B). These differences prevent existing methods from being used directly.

We propose a TinyCSI solver, which takes the profiles of these separated APIs as the input to the global joint optimization problem. Since sensing delay can easily affect the user experience, we take the overall time cost as the optimization criterion in our current design. Formally, TinyCSI solver solves an Integer Linear Programming (ILP) problem to generate the allocation results. The optimization objective can be formulated as:

$$Min \quad \sum_{i,r} x_{i,r} t_{i,r} + \sum_{j} \frac{F_{map}(x_j)}{w_{trans}}, \tag{1}$$

where

- $x_{i,r}$ denotes the i-th API in the callback function will be processed on computing resource $r \in \{node, server\}$.
- x_j indicates the j-th API which needs to be transmitted to other computing resources.
- $t_{i,r}$ denotes the time consumption for performing the i-th API on resource r.
- $F_{map}()$ is a linear function mapping the remotely executed API to the size of the application data (i.e. the input of the execution API) needed to be transmitted.
- w_{trans} is the most recently estimated transmission speed of the network in the unit of Kbps.

The objective expresses the estimated overall time consumption for executing these APIs on the assigned computing resources plus the time needed to transmit data for remote execution. The optimization process is also subject to the following hardware constraints (i.e., energy and memory limitation of nodes) and software constraints (i.e., continuity and concurrency of the system):

(1) Nodes are usually energy-limited. The total energy consumption for CSI collection, API execution and data transmission on the node must not exceed a threshold η:

$$s.t.\ p_{col}f+\sum_i x_{i,node}e_{i,node}+\sum_j F_{map}(x_j)e_{trans} \leq \eta, \quad (2)$$

where p_{col} is the estimated average power in mW for collecting a CSI measurement and f is the sample rate. $x_{i,node}$ denotes the i-th API will be processed on the node and $e_{i,node}$ denotes the measured energy consumption for executing the i-th API on the node. e_{trans} is the recently measured energy consumption of the node in mW for transmitting 1KB data to the network.

(2) Nodes usually have limited memory. The total memory cost on the node should also less than a threshold M:

$$s.t. \quad \sum_i x_{i,node}m_{i,node} \leq M, \quad (3)$$

where $m_{i,node}$ denotes the memory cost of the i-th API.

(3) Heavy tasks assigned on the node may not be handled in time due to its limited computational capability. When transmitting the input/output data, terrible network conditions will also affect the sensing delay. To ensure the continuity of the sensing system, the total execution time for processing APIs locally and remotely, as well as the data transmission time, should all be less than a time slot T:

$$s.t. \quad \forall r \sum_i x_{i,r}t_{i,r} \leq T, \quad \sum_j \frac{F_{map}(x_j)}{w_{trans}} \leq T. \quad (4)$$

(4) For each sensing task, the total number of APIs executed among *related* resources must be equal to the number of APIs N separated from user's callback function:

$$s.t. \quad \sum_r x_{i,r} = N. \quad (5)$$

The last two constraints also help build a pipeline for the streamed CSI data to deal with the input/output dependency between APIs. APIs from the same pipeline stage can run in parallel across different threads (depend on the allocated resources) to further improve system efficiency.

The above thresholds η, M, T need to be adjusted according to different sensing applications, whose user will certainly have different requirements. At present, we manually set these thresholds to empirical values for each application. Since automatic threshold adjustment based on user experience is another direction of work, we will consider it as possible future work of TinyCSI. It is also worth noting that our optimization process needs to be executed only once (unless the network transmission speed w_{trans} changes significantly) on the server for a sensing application. Therefore, we can utilize any heavy optimization algorithms in our TinyCSI solver to find the optimal scheduling solution. We finally select the standard optimization tool lp_solver [28] to solve the above optimization problem.

E. Standalone Mode

In contrast to the above two modes, the standalone mode allows sensing systems to run on individual nodes without connecting to a remote server. Since the sensing nodes usually have limited computing capacity, this mode is suitable for systems with low computation overhead. For most kinds of nodes (e.g. laptop, desktop) that support Matlab, the entire sensing system (except the GUI) can be directly deployed on them. For nodes (e.g. development board) that only support C, TinyCSI also provides two complete system migration approaches. The first approach is to let developers reproduce the callback function directly in C with the C-based TinyCSI library. In the meanwhile, the developers are required to set the CSI collection settings in the form of a JSON file manually. TinyCSI provides a JSON parser *JSONParser.exe* to help export the sensing configurations to the node. Then CSI data will be collected according to the settings and processed with the C-based callback. The second approach is to transfer the Matlab callback to C via Matlab Coder API [29] and the TinyCSI library. The transferred C-based callback has the same function signature and can be directly called by the node.

Tuning in the standalone mode is especially important since these standalone systems can be deployed in various environments. For nodes support Matlab, it is easy to turn back to the remote mode for parameter tuning. However, it is challenging for nodes only support C to tune/debug the sensing callback locally due to the lack of proper signal visualization support. To solve this problem, TinyCSI reuses MEX [23] in Matlab for tuning the C-based callback. Specifically, TinyCSI first transmits the C-based callback to the server. Then TinyCSI encapsulates the C-based callback as a MEX file and puts it into the SensingServer object (i.e., replacing *Callback.m* with the C-based callback). In this way, the streamed CSI data will be processed by the transmitted C-based callback with visualization support, which helps debug the C-based callback.

V. Implementation Details

In our developed demos, we encapsulate a HummingBoard Pro (HMB) mini-computer [30] (1.2GHz ARM Cortex-A9 processor and 1GB RAM) equipped with an off-the-shelf Intel 5300 NIC as the sensing node. Figure 3 shows the appearance and components of the sensing node, which is easy and feasible to be deployed anywhere. The CSI tool [1] is installed on each node for CSI collection. We use a commercially available router NETGEAR JR6100 as the AP. While TinyCSI can work well on any WiFi channel in both the 2.4 GHz/5 GHz bands, we use the 5 GHz band in our experiments because the wavelength of 5GHz is shorter and gives better resolution in sensing movement. We use a Dell desktop (3.2GHz i5 CPU and 8GB RAM) as the server.

Fig. 3: **Sensing node.**

Fig. 4: **Motion detection.**

Fig. 5: **Indoor localization.**

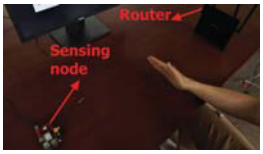

Fig. 6: **Gesture recognition.**

It is common to use multiple sensing nodes in a complex CSI-based sensing application. However, the lack of well-established socket support makes it challenging to realize the connection of multiple nodes. This is because Matlab's own socket library has two issues called randomly dropped packets and UI thread blocking, which is due to Matlab's single-thread nature [22]. We implement the Java socket interface shown in Figure 2 to support multi-threading. Specifically, a new thread will be automatically created via the interface when a new sensing node tries to connect to the server. For each thread, TinyCSI exports the transmitted CSI signals as a FIFO file and then process the file with Matlab in real-time. TinyCSI will also create a new GUI for visualizing the collected CSI data on each thread (i.e., each node).

VI. Demonstrative Systems

We have implemented three different types of CSI-based sensing demos with TinyCSI. These demos are chosen to cover the three major CSI-based sensing categories listed in Table I. Figure 4, 5, and 6 show the three representative demo systems deployed in the real world.

A. Demo System 1: Motion Detection

The first demo implemented with TinyCSI is a motion detection system, which continuously records the CSI measurements and detects whether there is an object moving nearby based on CSI variance. We chose this system to illustrate the basic steps of using TinyCSI for the simplicity of its signal processing procedure. It can also be easily extended to many other existing projects which sense the environments/objects based on the real-time processing of the CSI measurements.

In the remote mode, the motion detection system is implemented with two Matlab scripts, a main script *motionDetectionMain.m* and a callback script *motionDetectionCallback.m*. Figure 7 and Figure 8 show the implementation. As shown in Figure 7, *motionDetectionMain.m* is executed for the initialization of the system and CSI collection parameters, such as setting the server listening port and CSI sample rate. Some parameters necessary for the callback to process the CSI measurements, e.g., the detection threshold, can be assigned to a global variable *GP*. Values of the parameters in *GP* can later be updated in the GUI for fast tuning. Then the collected CSI data will be transmitted to the server via sockets and handled

```
% 1. Initialization Configurations
ServerPort = 8099;
SampleRate = 100;
Duration = 600;
StaticTime = [1, 10]; % Time for initialization
WindowLength = 10;

% 2. Global parameters to be tuned in the callback
global GP; % Parameters inside can be tuned with GUI
GP = struct();
GP.SampleRate = SampleRate;
GP.Duration = Duration;
GP.StaticTime = StaticTime;
GP.WindowLength = WindowLength;
GP.thres = 3;

% 3. Create SensingServer for calling the callback function
sensingserver = SensingServer(ServerPort,
        @motionDetectionCallback);
```

Fig. 7: **motionDetectionMain.m.**

```
function DetectionResult=motionDetectionCallback(context, action,
        CSIdata)
  global GP;
  Amplitude = GetAmplitude(1, 1, 15, CSIdata);
  PreprocessData = ButterworthFilter (Amplitude, 20, 5, 4,'low');

  % 1. Get CSI variance of static environments during initialization
  if action == context.CALLBACK_INIT
  StaticVar = GetVar(PreprocessData);

  % 2. Detect motion in real-time
  elseif action == context.CALLBACK_WINDOWDATA
  Result = GetVar(PreprocessData);
    if(Result/StaticVar > GP.thres)
        fprintf('Motion Detected!');
    else
        StaticVar = Result; % Update the static CSI variance
    end
  end
end
```

Fig. 8: **motionDetectionCallback.m.**

with the callback function *motionDetectionCallback*, which is repetitively called by the SensingServer object.

In the callback function, the assigned action argument can belong to 2 different types: CALLBACK_INIT and CALLBACK_WINDOWDATA. The CALLBACK_INIT action is taken to initialize necessary variables when the server is created. In this demo, we initialize the value of static CSI variance in different environments. The CALLBACK_WINDOWDATA action is taken to conduct the main processing algorithms of the callback function. In this demo, nearby moving object can be simply detected by comparing current CSI variance with the CSI variance pre-collected in a static room. As shown in Figure 8, we first extract amplitude information from the CSI data with *GetAmplitude* API. Next, a Butterworth filter, i.e. *ButterworthFilter*, can be directly applied to the received signals, and then the variance is calculated with a commonly used API *GetVar*. A motion event is detected once the ratio between current and static CSI variance *StaticVar* exceeds a predefined threshold *GP.thres*. While not detected, the static CSI variance should be updated to resist the interference of slight environmental changes.

We have implemented this demo in three scenarios: a meeting room, a corridor, and a lab. A volunteer is required

TABLE III: **Motion detection performance.**

Scenario	Meeting room	Corridor	Lab
Detection accuracy	93.3%	98.3%	94.7%
False alarm rate	3.7%	2.3%	2.0%

TABLE IV: **Indoor localization performance.**

Scenario	Meeting room	Corridor	Lab
Localization error (m)	0.67±0.16	0.99±0.57	0.88±0.21

to walk through the LoS path in the corridor and walk along an NLoS path 3m away from the wireless link in the meeting room and lab for motion detection. It is worth noting that *GP.thres* varies with different experimental environments and can be rapidly tuned with the GUI support. Table III shows the motion detection performance in the three scenarios. The overall detection accuracy is 95.4% and the overall false alarm rate is 2.7%. Among the three scenarios, the detection performance is relatively higher in the corridor due to the LoS path occupancy while the performance in the other two scenarios is still acceptable.

In the efficient mode, the callback function will be automatically separated into three TinyCSI APIs, i.e. *GetAmplitude*, *ButterworthFilter*, and *GetVar*. After the optimization, the three functions will be allocated to suitable places to minimize the detection delay. The details of the allocation results will be shown in Section VII. In the standalone mode, the simplicity of this callback makes it easy to be automatically transferred to the efficient C with the Matlab Coder API.

B. Demo System 2: Indoor Localization

The second demo is an indoor localization system with multiple nodes. This system consists of one AP and two nodes, which are placed at a height of 90cm. We use the classical MUSIC algorithm [31] to extract the AoA of the wireless path reflected by the target in each node for localization. With the two AoAs estimated from the two nodes, the target can be localized at the intersection of the two angles. Sensing the location via MUSIC AoA spectrum has been used to locate rogue Wi-Fi AP [32], improve Wi-Fi security [33], and track hand movement [34]. Our demo implemented with TinyCSI can be viewed as a generalization of these localization systems.

Most of our current implementation of this demo follows a similar pattern as in our previous demo. The largest difference is to initialize multiple threads in the SensingServer object for controlling multiple nodes to sense simultaneously. In this demo, we have a server that connects two nodes which both are responsible for sending the collected CSI measurements. In the main script, we configure two nodes getting connected and starting sensing simultaneously. There are also some minor changes including passing the coordinates of nodes and AP to the callback. In the callback function, we first extract the CSI data from each node. Then a *SubcarrierExtension* API will be called to transform the stacked CSI data to an extended CSI array, which increases the resolution of propagation paths [13]. Next, the callback utilizes a *MUSIC* API to calculate the AoA of each node. Combing the two AoAs, TinyCSI can estimate the location of the target. Table IV shows that this demo system can achieve an average median localization error of 85cm in typical indoor environments.

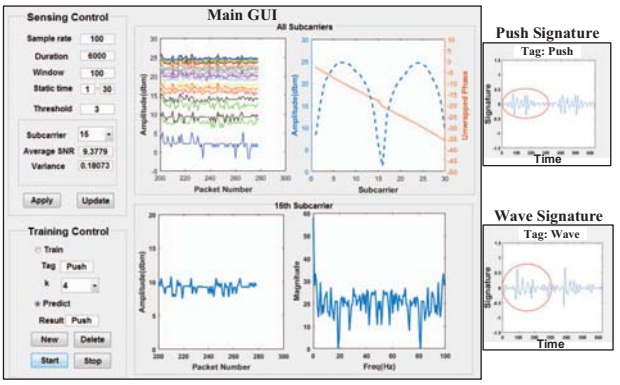

Fig. 9: **GUI of the gesture recognition demo.**

In the efficient mode of this demo, the computing resources include a server and two nodes. We use $r \in \{node\_1, node\_2, server\}$ to represent these resources in the optimization process. In addition, there are two sets of CSI-to-AoA functions (including *GetCSI*, *SubcarrierExtension*, and *MUSIC*) and a combination function *GetMean* to be allocated. The detailed allocation results will be shown in Section VII.

C. Demo System 3: Gesture Recognition

The third demo we developed is a gesture recognition system based on CSI signatures proposed in WiFinger [10]. Our demo can recognize four typical gestures that frequently occur in daily life: push, applaud, wave hand and turn hand over. This demo can be regarded as a generalization of existing CSI fingerprinting systems.

Referring to [10], we mainly use a k-NN classifier API *KNNClassifier* and a DTW distance calculation API *DTWDist* in the callback function for gesture recognition. In particular, we also integrated this demo into the GUI of TinyCSI. With this GUI, similar fingerprinting systems can be realized without even writing a single line of code. Figure 9 shows the GUI of this demo system built upon TinyCSI. After connecting the sensing node to the SensingServer, users can select the *Train* item on the *Training Control* panel and click the *New* button to create a new gesture tag for sensing. Then, users are asked to click the *Start* button and make the corresponding gesture for training data collection. After finishing the gesture, users should click the *Stop* button to save the corresponding CSI signature. The CSI signature of that gesture will be simultaneously shown in the GUI with the streamed CSI. Users can easily see whether they can obtain reliable and distinguishable CSI signatures of that gesture. Users can also adjust the essential parameter k of the k-NN classifier in the control panel. After training, users can select the *Predict* item, and click the *Start* button for testing data collection. Once the *Stop* button is clicked, the classification results will be shown in the GUI. We evaluated this demo in the above three scenarios. The AP and node are placed 1m apart and at a height of 90cm. The volunteer is asked to perform the above four gestures in the middle of the wireless link. In each scenario, we have collected 800 samples for each gesture and each sample lasts 10 seconds. We utilize 50% samples for training and the

TABLE V: **Gesture recognition accuracy in our demo.**

Gesture	Push	Applaud	Wave hand	Turn hand over
Meeting room	92.04%	87.25%	89.62%	86.13%
Corridor	92.22%	91.07%	90.58%	89.16%
Lab	90.42%	82.87%	90.72%	82.22%

TABLE VI: **The lines of code for implementing the three representative demos with and without TinyCSI.**

Demos	Motion detection	Indoor localization	Gesture recognition
Original implementation	Java 317 C 173 Matlab 132	Java 341 C 173 Matlab 198	Java 317 C 173 Matlab 261
With TinyCSI	Matlab 33	Matlab 55	Matlab 39
Reduction	94.7%	92.3%	94.8%

left 50% for testing. Table V shows the gesture recognition accuracy. Results show this demo is able to identify the above four gestures with an average accuracy of 88.7%, which is comparable with the experimental results in [10].

VII. Evaluation

In this section, we present the evaluation of TinyCSI. We evaluate the three demos in terms of ease of programming, improvement of sensing delay and implementation overhead.

A. Ease of Programming

Table VI shows the lines of code needed to implement the three representative demos between using TinyCSI and using the original APIs provided by Matlab. The Java code is implemented for the sever to send collection settings and control commands to the sensing node. For the indoor localization demo, we need to manually configure multiple connections to different nodes. The C code is implemented for transmitting the CSI data to the server in real-time and is the same across three demos. Results show that using TinyCSI reduces the lines of code by 93.9% on average. This is due to the appropriate API abstraction and the hidden development/connection details in TinyCSI.

B. Reduction of Sensing Delay

To evaluate the responsiveness of the efficient mode, we have measured the sensing delays of three working modes for the three representative demos. Sensing delay is defined as the time that putting a complete segment of CSI signals into the callback function to the time that getting the returned sensing results. In the remote and standalone mode, the sensing delay is the time consumption for executing the callback function, excluding the time for visualization. This is exactly the sensing delay of most existing CSI-based systems. We simply attach a pair of timestamps at the beginning and end of the callback function. In the efficient mode, the sensing delay includes the time for resource allocation, the time for resource transmission, and the time for all APIs to be completed.

To analyze the average sensing delay, we have collected 120s CSI data at a frequency of 100Hz for each demo. Data is transmitted through Wi-Fi and the median network throughput measured is around 1Mbps. We repeat the experiment 10 times. Table VII shows the sensing delays of three working modes in the three representative demos. Results show that: (1)

TABLE VII: **Sensing delays of the three working modes in the three representative demos (processing 120s CSI data).**

Demos	Standalone	Remote	Efficient
Motion detection	12.63±0.69s	8.60±0.75s	7.03±0.68s
Indoor localization	17.85±0.73s	13.26±0.71s	12.04±0.78s
Gesture recognition	21.58±1.57s	16.62±0.83s	13.94±0.78s

TABLE VIII: **Extra execution time and memory cost of the TinyCSI solver in the three representative demos.**

Demos	Motion detection	Indoor localization	Gesture recognition
Time (s)	1.34±0.36	2.70±0.40	2.42±0.38
Memory (MB)	11.32±1.07	14.58±0.58	14.61±0.82

For the motion detection demo, its *GetAmplitude* function is allocated to the node while *ButterworthFilter* and *GetVar* will run on the server. The efficient mode reduces the sensing delay by 44% and 18% relative to the standalone mode and remote mode, respectively. (2) For the indoor localization demo, the two *GetCSI* APIs and two *SubcarrierExtension* APIs are respectively allocated to the two nodes and the remaining three APIs are allocated to the server. The efficient mode reduces the sensing delay by 33% and 9% relative to the standalone mode and remote mode, respectively. (3) For the gesture recognition demo, the first two APIs (i.e. *GetAmplitude*, *ButterworthFilter*) are allocated to the node while the other five APIs are allocated to the server. The efficient mode reduces the sensing delay by 35% and 16% relative to the standalone mode and remote mode, respectively. The low sensing delays of the efficient mode indicate that this mode can significantly improve the system responsiveness with the TinyCSI solver. Results also show that simple signal processing at the start of the callback function is more suitable to be executed on the nodes.

C. Implementation Overhead

Execution Time Overhead. While the efficient mode significantly improves the system responsiveness, its optimization process can also introduce non-negligible execution time overhead. Table VIII shows the execution time of the TinyCSI solver in our demos. Results show that the execution time of the solver is largely determined by the number of APIs to be managed and the number of computing resources. A larger number of APIs and computing resources will consume a longer optimization time. However, these few seconds of optimization time are acceptable since the TinyCSI solver executes the optimization process only once when the system starts running or the network condition changes significantly.

Memory Overhead. Running the SensingServer object introduces extra memory cost to the server. We evaluate the average memory cost of each component in *SensingServer.m* when running the three representative demos in the remote mode. We need approximately 5.4MB of memory to run the *ControlServer.jar*. The memory overhead for the *GUI.m* is within 13MB. For FIFO files, the overheads of writing and reading at a rate of 100Hz are 3.8MB and 4.3MB, respectively. Moreover, as shown in Table VIII, the average memory costs of the optimization process in the three demos are 11.32MB, 14.58MB and 14.61MB, respectively. As a result, the memory overhead of TinyCSI is negligible for a common server.

TABLE IX: **Average sensing delays of the three representative systems developed by users.**

Systems	Original (remote mode)	Efficient mode
Keystroke recognition	0.116s	0.096s
Target tracking	0.171s	0.156s
Activity classification	0.153s	0.128s

D. User Study

To evaluate the applicability of TinyCSI, we conducted user studies and chose three representative users (including experienced and novice developers) for analysis. These users develop a keystroke recognition system, a target tracking system, and an activity classification system using TinyCSI, respectively. At the end of their development processes, we also did a survey to evaluate their user experience of TinyCSI.

The biggest benefit reported by the three users is the remote mode. With the well-abstracted API library and GUI, they can intuitively understand how the CSI signals/features change when they deploy their systems with different algorithms/parameters at different places. In addition, the efficient mode does benefit their sensing systems. Table IX shows that our efficient mode can effectively reduce the sensing delay by 8.8%-17.2% for their systems.

VIII. Conclusion

In this paper we present TinyCSI, a rapid development framework for CSI-based sensing applications. Developers only need to write a main script to determine the CSI collection settings and a callback function with our predefined Matlab/C-based library to realize the essential sensing algorithms. TinyCSI provides a remote mode for developers to rapidly validate algorithms and related parameters, an efficient mode for achieving a high system sensing responsiveness, and a standalone mode for running systems without network access. We implemented TinyCSI and evaluated its performance using three representative demos and real-world user studies. Results show that TinyCSI helps significantly reduce the development efforts and sensing delay, while incurring acceptable overhead.

Acknowledgment: This work is supported by the National Key R&D Program of China under Grant No. 2019YFB1600700 and the National Science Foundation of China (No. 61872437). Wei Dong is the corresponding author.

References

[1] D. Halperin, W. Hu, A. Sheth, and D. Wetherall, "Tool release: Gathering 802.11n traces with channel state information," *ACM SIGCOMM CCR*, vol. 41, no. 1, p. 53, Jan. 2011.

[2] Y. Xie, Z. Li, and M. Li, "Precise power delay profiling with commodity wifi," in *Proceedings of ACM MobiCom*, 2015.

[3] Z. Zhou, Z. Yang, C. Wu, L. Shangguan, and Y. Liu, "Towards omnidirectional passive human detection," in *Proc. of IEEE INFOCOM*, 2013, pp. 3057–3065.

[4] J. Lv, W. Yang, L. Gong, D. Man, and X. Du, "Robust wlan-based indoor fine-grained intrusion detection," in *Proc. of IEEE GLOBECOM*, 2016, pp. 1–6.

[5] C. Wu, Z. Yang, Z. Zhou, X. Liu, Y. Liu, and J. Cao, "Non-invasive detection of moving and stationary human with wifi," *IEEE JSAC*, vol. 33, no. 11, pp. 2329–2342, 2015.

[6] K. Qian, C. Wu, Z. Yang, Y. Liu, and Z. Zhou, "Pads: Passive detection of moving targets with dynamic speed using phy layer information," in *Proc. of IEEE ICPADS*, 2014, pp. 1–8.

[7] J. Xiao, K. Wu, Y. Yi, L. Wang, and L. M. Ni, "Fimd: Fine-grained device-free motion detection," in *Proc. of IEEE ICPADS*, 2012.

[8] W. Wang, A. X. Liu, M. Shahzad, K. Ling, and S. Lu, "Understanding and modeling of wifi signal based human activity recognition," in *Proc. of ACM MobiCom*, 2015, pp. 65–76.

[9] B. Fang, N. D. Lane, M. Zhang, A. Boran, and F. Kawsar, "Bodyscan: Enabling radio-based sensing on wearable devices for contactless activity and vital sign monitoring," in *Proc. of ACM MobiSys*, 2016.

[10] H. Li, W. Yang, J. Wang, Y. Xu, and L. Huang, "Wifinger: talk to your smart devices with finger-grained gesture," in *Proc. of ACM UbiComp*, 2016, pp. 250–261.

[11] Y. Ma, G. Zhou, S. Wang, H. Zhao, and W. Jung, "Signfi: Sign language recognition using wifi," *Proc. of ACM IMWUT*, vol. 2, no. 1, p. 23, 2018.

[12] X. Li, S. Li, D. Zhang, J. Xiong, Y. Wang, and H. Mei, "Dynamic-music: accurate device-free indoor localization," in *Proc. of ACM UbiComp*, 2016, pp. 196–207.

[13] M. Kotaru, K. Joshi, D. Bharadia, and S. Katti, "Spotfi: Decimeter level localization using wifi," in *ACM SIGCOMM CCR*, vol. 45, no. 4, 2015.

[14] K. Wu, J. Xiao, Y. Yi, M. Gao, and L. M. Ni, "Fila: Fine-grained indoor localization," in *Proc. of IEEE INFOCOM*, 2012, pp. 2210–2218.

[15] J. Xiao, K. Wu, Y. Yi, L. Wang, and L. M. Ni, "Pilot: Passive device-free indoor localization using channel state information," in *Proc. of IEEE ICDCS*, 2013, pp. 236–245.

[16] J. Wang, H. Jiang, J. Xiong, K. Jamieson, X. Chen, D. Fang, and B. Xie, "Lifs: low human-effort, device-free localization with fine-grained subcarrier information," in *Proc. of ACM MobiCom*, 2016.

[17] A. Virmani and M. Shahzad, "Position and orientation agnostic gesture recognition using wifi," in *Proc. of ACM MobiSys*, 2017, pp. 252–264.

[18] *IEEE 802.11n-2009-Amendment 5: Enhancements for Higher Throughput*. IEEE-SA, 2009.

[19] G. Guan, W. Dong, Y. Gao, K. Fu, and Z. Cheng, "Tinylink: A holistic system for rapid development of iot applications," in *Proc. of ACM MobiCom*, 2017, pp. 383–395.

[20] L. Ravindranath, A. Thiagarajan, H. Balakrishnan, and S. Madden, "Code in the air: simplifying sensing and coordination tasks on smartphones," in *Proc. of the Twelfth Workshop on Mobile Computing Systems & Applications*. ACM, 2012, p. 4.

[21] S. Elmalaki, L. Wanner, and M. Srivastava, "Caredroid: Adaptation framework for android context-aware applications," in *Proc. of ACM MobiCom*, 2015, pp. 386–399.

[22] Y.-C. Tung, D. Bui, and K. G. Shin, "Cross-platform support for rapid development of mobile acoustic sensing applications," in *Proc. of ACM MobiSys*, 2018, pp. 455–467.

[23] MEX, "Build MEX function from C/C++ or Fortran source code," https://ww2.mathworks.cn/help/matlab/ref/mex.html.

[24] M. Shahzad and S. Zhang, "Augmenting user identification with wifi based gesture recognition," *Proc. of ACM IMWUT*, vol. 2, no. 3, p. 134, 2018.

[25] C.-C. Chang and C.-J. Lin, "LIBSVM: A library for support vector machines," *ACM Transactions on Intelligent Systems and Technology*, vol. 2, pp. 27:1–27:27, 2011.

[26] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, "Maui: making smartphones last longer with code offload," in *Proc. of ACM MobiSys*, 2010, pp. 49–62.

[27] P. Georgiev, N. D. Lane, K. K. Rachuri, and C. Mascolo, "Leo: Scheduling sensor inference algorithms across heterogeneous mobile processors and network resources," in *Proc. of ACM MobiCom*, 2016.

[28] M. Berkelaar, K. Eikland, and P. Notebaert, "lp_solve 5.5, open source (mixed-integer) linear programming system. software, may 1 2004."

[29] "Matlab coder app," https://www.mathworks.com/products/matlab-coder/apps.

[30] SolidRun, "HummingBoard Pro," http://wiki.solid-run.com/doku.php?id=products:imx6:hummingboard, 2014.

[31] R. Schmidt, "Multiple emitter location and signal parameter estimation," *IEEE T ANTENN PROPAG*, vol. 34, no. 3, pp. 276–280, 1986.

[32] A. Tzur, O. Amrani, and A. Wool, "Direction finding of rogue wi-fi access points using an off-the-shelf mimo–ofdm receiver," *Physical Communication*, vol. 17, pp. 149–164, 2015.

[33] J. Xiong and K. Jamieson, "Securearray: Improving wifi security with fine-grained physical-layer information," in *Proc. of ACM MobiCom*, 2013, pp. 441–452.

[34] J. Zhu, Y. Im, S. Mishra, and S. Ha, "Calibrating time-variant, device-specific phase noise for cots wifi devices," in *Proc. of ACM SenSys*, 2017, p. 15.