B-DTN7: Browser-based Disruption-tolerant Networking via Bundle Protocol 7

Lars Baumgärtner*, Jonas Höchst†, Tobias Meuser‡
* Technische Universität Darmstadt, FB 20, D-64289 Darmstadt, Germany
E-mail: baumgaertner@cs.tu-darmstadt.de
†Philipps-Universität Marburg, FB 12, D-35032 Marburg, Germany
E-mail: hoechst@informatik.uni-marburg.de
‡ Multimedia Communicaations Lab, Technische Universität Darmstadt, D-64289 Darmstadt, Germany
E-mail: tobias.meuser@kom.tu-darmstadt.de

***Abstract*—During and after the event of a crisis, means of communication are vital for civilians and professional responders alike. In particular, technologies such as disruption-tolerant networking (DTN) can play a key role to distribute data even under challenging network conditions. Such networks benefit from an increased number of mobile participants since they help distribute messages to remote places. Unfortunately, it is unrealistic to assume that smartphone vendors will ship the necessary software or users will be prepared with the needed apps installed before the crisis happens. Currently, smartphone apps usually need Internet connections to be installed from an app store and/or are very platform-specific, i.e., there is no general device-to-device app distribution. We present a novel solution to let any user with a web browser persistently participate in a DTN network. By leveraging a newly written Bundle Protocol 7 draft implementation in the programming language Rust and deploying it to WebAssembly, we provide a secure and efficient way for backend and frontend DTN networks. The presented solution incorporates classic DTN daemons and access points for web app distribution and bundle synchronisation. Through benchmarks, we show the efficient processing of bundles and the feasibility of bundle handling in browsers. All code is available as open-source under a permissive license.**

I. Introduction

Disruption-tolerant networking is used in many different scenarios such as Interplanetary Networking, wildlife monitoring, and emergency communication. While protocols constantly evolve, e.g., the official draft of DTN bundle protocol version 7 [1], some problems for mass adoption in specific scenarios are still not solved satisfyingly. To enable DTN communication for regular users during events such as demonstrations, festivals, or a crisis, where the communication infrastructure is often disrupted, people usually have to be prepared and install the necessary communication software, which normally needs to happen before the crisis due to the various security mechanisms on current smartphone operating systems, which often only permit installation of signed apps via an official app store. Modern technologies like WebAssembly (wasm) and the web platform in general can be used to provide a zero-installation, offline way for people to join the communication only using a web browser, which is readily available on all modern devices. Furthermore, having people roam various wireless networks while participating in the DTN also increases delivery rates as users act as data mules to connect separated networks. Ideally, one should also be able to have other ways of transmitting DTN bundles, even when no radio link is currently available.

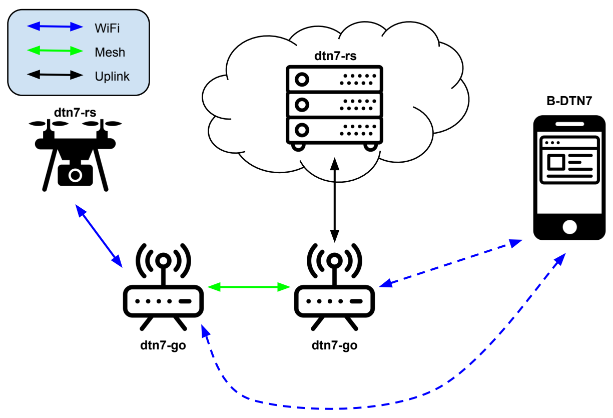


Fig. 1. Architecture of our communication infrastructure

Our approach is based upon a novel implementation of the most recent bundle protocol 7 draft and the MTCP convergence layer [2] written in the Rust programming language. We separated the pure bundle handling parts described in the RFC from the network-related ones and the general agent functionality. Therefore, a minimal core for target platforms with limited capabilities is directly available. Furthermore, these parts are then used via WebAssembly to write a progressive web app that can handle bundles and work offline in any browser using HTML5 offline storage. As a bridge between classic DTN networks and browser-based clients, a thin web server component was designed to serve bundles as well as the app itself. By having public hot spots such as the ones provided by Freifunk or in streetlights of smart cities, everyone can easily participate in the communication even without proper internet uplink. Figure 1 depicts such a scenario where we have a heterogeneous DTN network between meshed routers, wired internet uplink, and WiFi participants such as drones, all running different dtn7 implementations. The WiFi participants also have the possibility to generate data, e.g., using their local sensors. Also shown there, smartphones with our progressive web app that can synchronize with any access point in range and carry on data to other networks. Besides relying on existing WiFi infrastructure, we also provide alternative sync mechanisms that directly transfer bundles via QR codes or through (ultrasonic) sound from one device to another. Especially broadcasting via audio can be used for 1:n communication from local devices or through public speakers to distribute public data, e.g., maps, instructions, and announcements.

Our evaluation shows that the Rust implementation and wasm as a platform provide enough performance, outperforming other native implementations, for even higher bundle workloads, while at the same time leaving only a minimal memory/storage footprint. The practical evaluation of B-DTN7 shows the general usefulness of our approach and also the limitations regarding the alternative transmission methods.

In summary, we make the following contributions:

- A novel and secure bundle agent based on the most recent Bundle Protocol Version 7 draft written in Rust[1]
- A lean, fast and portable bundle protocol implementation[2]
- A novel, modern web app that can turn any browser into a DTN participant
- A novel bridging server connecting bundle agents with web clients and delivering the web app.
- A unique convergence layer using QR codes to transmit bundles offline directly from device to device
- A novel way to distribute bundles via (ultrasonic) sound from one device to multiple others nearby via the web app

The paper is organized as follows. Section II discusses related work. In Section III, we present requirements and designs decisions. Section IV discusses implementation issues. Section V presents experimental results. Section VI concludes the paper and outlines areas of future work.

II. Related Work

Since the advent of disruption-tolerant networking, it has been studied in different forms and aspects. A reference implementation of the current standard [3], called *DTN2* [4], provided by the IETF DTN working group exists. For the Interplanetary Overlay Network (ION) developed by NASA, the focus is set to extreme distances and predicable connections losses in space [5]. *IBR-DTN* [6] is a DTN software for terrestrial use, offering various example applications and thus real-world applicability. The more recent Bundle Protocol 7 (BP7) is implemented in Micro Planetary Communication Network (*µPCN*) and designed to connect different regions of the world [7]. Terra is a Java implementation of BP7, claiming to be lightweight and modular [8]. *DTN7* is a BP7 implementation in Golang, offering a modular, extendable and portable implementation, as well as high performance in competitive network evaluations [9]. Except for the last one, all other implementations are based on older drafts of BP7 and, thus, incompatible.

[1]https://crates.io/crates/dtn7
[2]https://crates.io/crates/bp7

Serval is a mesh networking application which is also available on mobile platforms. It features its own delay-tolerant component called Rhizome which offers different convergence layers, such as packet radio, serial ports, WiFi, Bluetooth or HTTP-based bundle submission [10]. *Forban* is a link-local peer-to-peer software. It runs as a regular program and implements its bundle exchange mechanism on top of HTTP [11], which would in principle be extendable to a browser-based bundle exchange. *FireChat* is a mobile application and uses multiple platform-specific and generic communication protocols, such as Apple Wireless Direct Link (AWDL), Bluetooth or WiFi Direct to transmit messages [12].

There exist convergence layer implementations for IBR-DTN based on HTTP and 802.15.4 (ZigBee), specifically targeting sensor networks [13]. AX.25 packet radio is targeted as a convergence layer by the work of Ronan et al. [14]. *TXQR* is a library that uses animated QR codes and forward error correction (FEC) to transmit data between devices using their display and camera [15] reaching up to 200 kbps by using FEC with fountain codes [16]. *Quiet* is a framework designed to transmit data via audible or inaudible sound samples [17]. It offers various profiles targeted for transmission via speakers and microphones or via a direct cable connection, allowing transmission speeds of up to 40 kbps.

In *EmerGence* a progressive web app served by unmanned aerial vehicles is proposed to enable communication in emergency or natural disaster situations [18]. Sankaran et al. developed a framework for highly-localized mobile web applications, delivered via DTN. While the DTN software itself does not run inside of the browser, specific applications and the communication of those can use DTN from inside the browser [19].

Disruption-tolerant networks were shown to be a resilient communication-enabler in emergency communications and challenged networks [20], [21]. Particularly in combination with sensor networks DTNs show promising results [22], [23]. Other use cases of DTNs can be found in enabling communications for rural areas [24], [25].

III. Design

As our goal is to integrate as many users as possible in our DTN, we need to interoperate with existing Bundle Protocol Version 7 draft [1] implementations such as *dtn7-go* but on the other hand enable users to easily participate in the communication. The latter is especially difficult if the process of submitting native apps to various app stores needs to be avoided. Also, in emergency scenarios, people might not be able to install new apps, and only a minor share of the people might have installed the needed apps before the emergency case. Thus, progressive web apps that also work offline can be used as an alternative to typical mobile applications. Having these mobile devices in the network benefits not only the users themselves but also for other participants, as every user acts as a data mule and can distribute data to otherwise unconnected and isolated network segments.

All relevant components for our approach can be seen in Figure 2. The most basic requirement is an implementation of

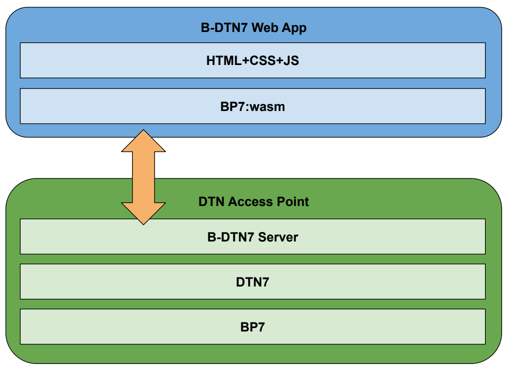


Fig. 2. Architecture of Software Stack

the bundle protocol itself, as this is used in the backend as well as in the web app. Furthermore, the DTN access point must contain a *dtn7* capable bundle agent as well as a bridging server to deliver the web app and enable synchronisation. The web app itself must utilize the *bp7* library and should build on standard web technologies such as wasm, HTML, CSS, and JavaScript.

A. Bundle Protocol

The new Bundle Protocol Version 7 draft has some significant changes compared to RFC 5050 [3]. Most prominently, encoding is performed through standard CBOR encoding [26]. Furthermore, configurable CRC checksums were added, as some convergence layers might not be reliable. Bundle compilation was also simplified, as bundles always begin with a Primary Block. Other functionality, e.g., routing information, signatures or other metadata can be realized using Canonical Blocks.

BP7 defines the Bundle Protocol Agent (BPA), which is responsible for communication between the other components and, e.g., creates bundles. An Application Agent (AA) is responsible to interconnect DTN applications to the BPA. Convergence Layers Adapters (CLA) implement Convergence Layers (CL), which are used to establish connections between nodes. CLs can be of different types, bi- or unidirectional, slow (LoRa) or fast (WiFi) and of different qualities, e.g., long-lived or ephemeral.

While the bundle protocol 7 draft contains information about encoding individual bundles as well as processing and delivering them, it is a reasonable decision to split this functionality into two separate libraries. As the encoding can easily be ported to and used in embedded devices, the minimal needed functionality should be compiled in a bundle protocol library (*bp7*). Everything else needed for a complete bundle protocol agent including application agents, routing and convergence layers should be build upon this lib in another project (*dtn7*). The *bp7* lib should also be reusable as a WebAssembly library within the browser. The complete software stack can be seen in Figure 2.

B. B-DTN7 Server

A new gateway software is needed to bridge between classic DTN networks and web-based clients. This server must serve the B-DTN7 web app as well as synchronize bundles with multiple browser B-DTN7 instances. Following the Unix philosophy we chose to build a separate server specifically designed for handling this task and interacting as a client to a local *dtn7* daemon. This way the underlying *dtn7* daemon implementation can be swapped out, and it makes the whole project more maintainable in the long run. The web app should be served as static web content and the necessary synchronisation functions for, e.g., getting a list of known bundles or adding new bundles, should be implemented as REST calls.

C. B-DTN7 Web App

The main component of the web app should be the *bp7* library that is also used for the native *dtn7* daemon. Through the use of the HTML5 local storage facilities, it is possible to store bundles persistently on the browser device for later use. Furthermore, besides just syncing existing bundles, it should be possible to create new bundles with text messages for emergency communication. If not even local networks are available, the last resort feature should be to pass selected messages directly from device-to-device via QR code sharing. Due to size limitations of QR codes, this can only be used for rather small bundles, but the compression of bundles can help to stretch these boundaries. Besides making cameras accessible from web pages, HTML5 also provides easy access to the microphone and speakers. Thus, modulated sound can be sent from one device to be received by others and demodulated again, resulting in another offline communication channel usable for larger bundles.

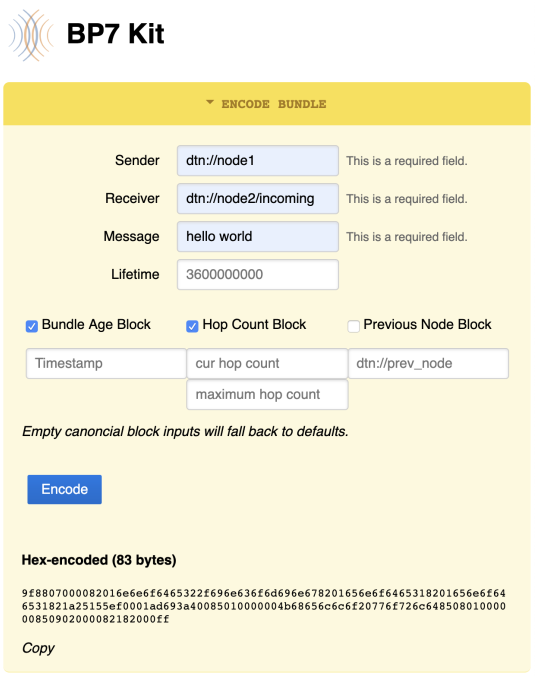
IV. Implementation

A. Bundle Protocol

As one of the design goals is to target small devices, efficiency and a low resource footprint heavily influence the programming language used. Furthermore, security is of utmost concern. Thus, memory safety and the avoidance of undefined behaviour must also be considered. This leads us to the Rust programming language, which delivers performance comparable to C++ but makes strong safety guarantees and, through *llvm*, can even target embedded systems or output WebAssembly directly. Besides being available as a library, a command-line tool is shipped to debug, encode and decode bundles in various forms. Furthermore, building upon the wasm output a web-based helper is provided to generate bundles for testing as shown in Figure 3.

B. B-DTN7 Server

This component serves multiple purposes. First, it must serve the web app to mobile users via HTTP(S). Second, it must provide a REST interface for synchronizing bundles with the web apps. Third, it must interact with a local *dtn7* daemon to forward bundles between web-connected users and bundle



Sender: dtn://node1 — This is a required field.
Receiver: dtn://node2/incoming — This is a required field.
Message: hello world — This is a required field.
Lifetime: 3600000000

[✓] Bundle Age Block [✓] Hop Count Block [ ] Previous Node Block

Timestamp | cur hop count | dtn://prev_node
maximum hop count

Empty canoncial block inputs will fall back to defaults.

Hex-encoded (83 bytes)

```
9f88070000820l6e6e6f6465322f696e636f6d696e678201656e6f6465318201656e6f64
6531821a25155ef0001ad693a40085010000004b68656c6c6f20776f726c648508010000
0085090200008218200ff
```

Fig. 3. Web helper tool shipped with BP7 to create custom bundles.

agents using classic convergence layers such as MTCP. As this component is heavily based upon web technologies and not performance critical, the proof-of-concept implementation was realized using Google's Go programming language - as it contains all necessary functionality in its standard library. This server is meant to be run on the network edge at access points or as part of a captive portal. Therefore, the app can be distributed even without a working internet connection.

C. B-DTN7 Web App

Since Rust can compile to WebAssembly, the *bp7* library can easily be reused in the web app as well. To rapidly prototype B-DTN7, we choose JavaScript. Therefore, it is possible to access the HTML5 storage and to request further resources from the server, such as new bundles. The whole app consists only of a few files that can be served statically. In the future, all logic could be implemented as a wasm module, eliminating JavaScript almost completely. As the web app follows responsive design principles, it can adapt to various devices such as desktop browsers, tablets, and smart phones.

Due to the browsers security restrictions, e.g., same origin-policy, it is not easily possible to implement peer-to-peer communication in the web app. Thus, bundle synchronisation by default happens with the IP address from which the app was served. In case of large mesh communities such as Freifunk or GuiFi this would also be the access point through which the mobile device is connected to the WiFi. When acting as a captive portal, all users can easily be made aware of the service and directly participate. In the future, a combination of WebRTC and QR codes for signalling data transfer could be used for an actual web-based peer-to-peer system without a fixed third party server[3]. To support direct bundle transfer without any network connectivity whatsoever, the app can present any small bundle encoded as a QR code. Which in return can be scanned and imported on another device running B-DTN7. Due to the limited number of bytes that can be packed into a QR code, all bundles are compressed using a LZ-based algorithm. This heavily reduces the number of bytes needed to be transmitted, e.g., a benchmark bundle of 281 bytes can be transmitted in only 72 encoded and compressed bytes in the QR code. As the local HTML5 Storage only supports a maximum size of 5 MB worth of string data to be stored offline, this compression can also be used to store more bundles locally. The possibility for bundle transmission via audio depends heavily on the used browser and security settings. Prior to transmission via audio, we compress the bundle again and then transmit it as a hex-encoded string. Depending on whether two devices are connected via an audio cable or the audio is transmitted over the air, various transmission profiles can be used. We can even broadcast bundles via ultrasonic sound, which makes it is almost unnoticeable for humans.

V. Experimental Evaluation

To evaluate B-DTN7, we conducted several tests, ranging from pure benchmarks to real-world deployments. While the benchmarks can give a general impression of the quality and usefulness of our Rust implementation of the protocol draft, the real-world deployment case study shows the practicality and the limitations of our approach and the overall system.

A. Rust Bundle Protocol 7 Evaluation

To evaluate our Rust implementation of the bundle protocol 7 draft, we mainly focus on the encoding and decoding aspects of the implementation in different environments as this is most relevant for our application of Bundles-in-Browser. Due to the fact that the protocol draft is rather new, there are not many alternative implementations. Thus, we focus on comparison with *dtn7-go*, which is quite up-to-date and can also be deployed to WebAssembly. Here, the evaluation mainly focuses on the recent stable version 0.2 but to explain some effects the previous version 0.1 was also included for some additional benchmarks as major critical parts of *dtn7-go* have been rewritten between these two versions. For the Rust implementation, the most recent version, 0.3.7, was evaluated. The whole evaluation setup is available as a docker container[4] and the raw results can be downloaded[5] for further evaluation as well. The hardware used for evaluation is a Quad Core 2.3 GHz Intel Core i5 with 16 GB RAM. If not stated otherwise, browser based benchmarks were performed in Google's Chrome.

1) Performance: The core metrics that are relevant for classic bundle agents, routers, and web clients alike is the raw processing speed of bundles. More specifically, how fast can new bundles be created, existing bundles encoded and finally how fast can they be decoded and parsed again. For these

[3]https://github.com/AquiGorka/webrtc-qr
[4]https://github.com/stg-tud/bp7eval
[5]dat://bdtn7raw.hashbase.io

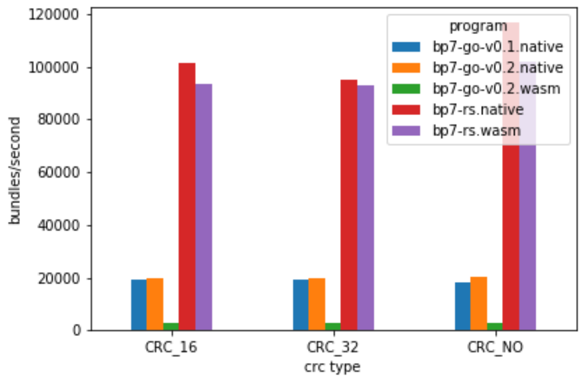


Fig. 4. Bundle creation performance

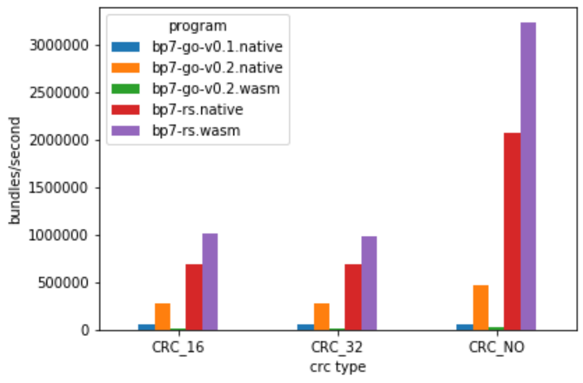


Fig. 5. Bundle encoding performance

benchmarks standard bundles with a small 3 byte payload and two extension blocks (bundle age and hop count) are created. All tests were performed without CRC checksum, with CRC16 enabled and with CRC32 enabled to identify possible overhead through checksum calculation. The benchmarks ran 3 times plus one additional warm-up run beforehand.

a) Bundle Creation: Of the three performance evaluation subjects, the bundle creation is the operation usually performed least often. Complete bundles are only created when new content is to be distributed or a new administrative record has to be sent. Thus, its performance is not as critical as pure encoding and decoding operations. Nevertheless, when used on small sensor nodes or energy-constrained devices, the efficiency of the bundle creation mechanisms is pivotal for the performance of these.

The results of the bundle creation benchmarks can be seen in Figure 4. Due to the fact that bundle creation involves many allocations and other calculations, the effects of the used checksum settings are mostly neglectable. This is also consistent across all implementations. One can clearly see that the Rust implementations are significantly faster than the Go version, around 100.000 bundles/second for Rust versus less than 20.000 for the native Go version. In case of Go wasm, there is even a speed difference up to two orders of magnitude. When comparing the native Rust version with the wasm one, the native one has a slight edge probably due to the easier and more direct memory management. Also, as timestamps are added during the bundle creation process, this requires a bridged JavaScript call in wasm to get the current time, which adds extra overhead compared to the direct system call in the native version.

b) Bundle Encoding: The pure encoding speed is especially relevant for less capable devices such as small sensor nodes and central routers that need to handle large numbers of bundles. Each bundle needs to be re-encoded before transmission to update hop counts, previous node blocks and/or bundle age blocks. Thus, encoding is a frequently performed operation in environments with many participants and a high bundle load.

To prevent any unfair JIT optimizations in wasm the previously encoded bundle's lifetime field is increased before encoding. As shown in Figure 5, the raw encoding performance is 5 to 10 times higher than the bundle creation performance. This performance increase can easily be explained with the fact that only the CBOR encoding has to be performed and the optional checksum has to be calculated. Here, the variants without CRC checksum are significantly (factor 2 or more) faster than the CRC16 and CRC32 variants across all implementations. Also noteworthy that the runtime difference between the CRC16 and CRC32 variants are neglectable. So if slightly larger bundles sizes are not an issue, CRC32 is to be preferred. The Rust implementation, depending on the configuration, can encode from 500k bundles/second up to over 3 million bundles per second. The Go native version in average has problems reaching more than 200k bundles/second and in case of the wasm version is way below 10.000 bundles/second. Interestingly enough, the Rust wasm version is even faster than its native counterpart. This could be related to JIT optimizations or that the data structures and calculations can be handled very effectively in wasm, not needing any bridges to JavaScript functions or any external functionality.

c) Bundle Decoding: Similarly to bundle encoding the decoding process is a vital operation that is performed quite often by any involved bundle agent. Each bundle received or loaded from disk has to be decoded first, which also includes verifying CRC checksum and performing various validity checks.

The results of this evaluation are shown in Figure 6. While decoding is significantly slower than pure encoding it is slightly faster than bundle creation. Again, the differences between different checksum settings are mostly neglectable. Here, the Rust versions are in the order of one or two magnitudes faster than the Go implementations, 500-800k versus less than 10-50k. Just as seen in the bundle creation benchmarks the native version outperforms the wasm Rust version again. Since decoding a bundle means recreating bundle structures this might also be a result of the better memory management in the native version.

d) Performance Conclusion: Across all benchmarks one can clearly see that the Rust version is consistently at least 5 to 10 times faster than the Go version. This is especially relevant for nodes handling many bundles or very constraint devices. When targeting wasm as a platform the very poor performance of the Go version is a real show stopper and the

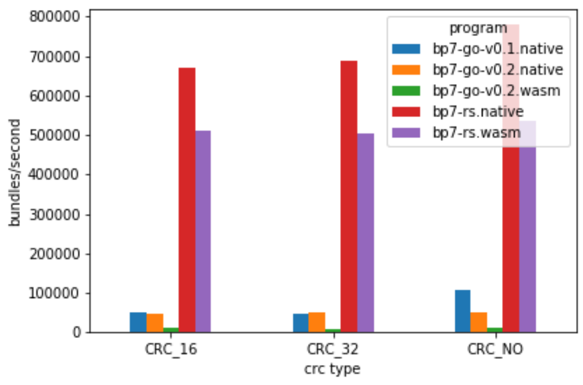


Fig. 6. Bundle decoding performance

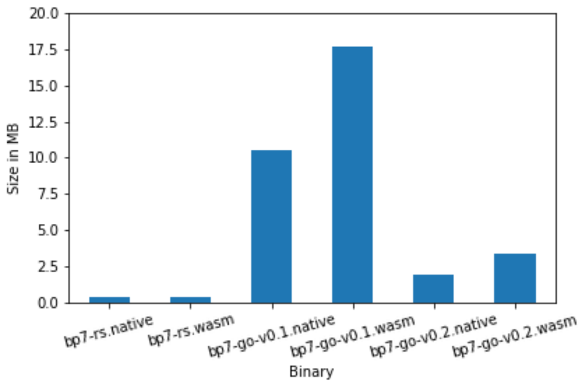


Fig. 7. Binary sizes of the various benchmarks

benefits of the Rust implementation come into play.

2) Binary Sizes: Binary sizes might not be so relevant for laptops, cloud instances or even Raspberry Pi grade SBCs but when targeting embedded devices or even the web platform, one needs to keep the binary footprint low for efficiency. For this evaluation all native binaries have been previously stripped[6] and for the wasm parts only the assembly file itself was measured, while the generated JavaScript glue code with a size of less than 25 KB is neglectable.

The different binary sizes are shown in Figure 7. A big factor influencing the binary sizes are used dependencies, which also explains the sizes of the Go implementation v0.1 ranging from 11 to 17 MB. Many of these dependencies have been removed in the now current version 0.2 which is significantly smaller with about 3 MB. Still the stripped Rust binaries are significantly smaller with less than 0.5 MB. Especially for web apps, smaller code sizes are relevant, since they directly influence load times, bandwidth usage and the user experience. Also the memory accessible for each wasm app in a browser tab is rather limited.

B. B-DTN7 in Action

To evaluate our approach under real world circumstances we set up a small network consisting of two access points, a

[6]http://pubs.opengroup.org/onlinepubs/9699919799/utilities/strip.html

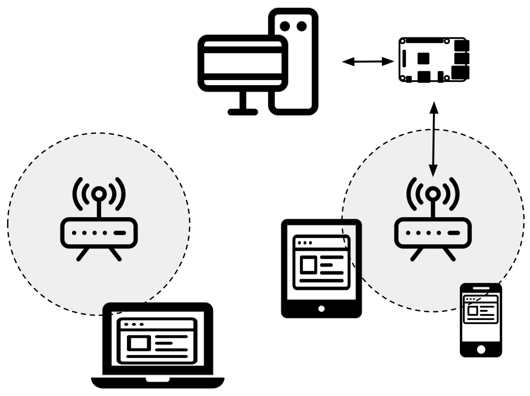


Fig. 8. Network setup of real life evaluation.

backend router, a wired workstation and three different mobile devices. The complete setup is visualized in Figure 8. One access point is isolated and dependant on roaming users to synchronize bundles. There is one user permanently connected to this access point and injecting new bundles there. Connected to the second access point is a network with a DTN router and a wired workstation also producing new content. Mobile clients used in this case study are laptops, tablet computers and smartphones. B-DTN7 was tested on iOS and Android based devices with their corresponding default browsers as well as Firefox and Google Chrome on Linux and macOS desktops.

1) Browser performance: Besides limited connectivity options, the biggest constraint of a browser-based DTN agent is the storage capacity available. In our tests, we were able to fit around 16.000 of our benchmark bundles into the 5 MB HTML storage available to each web app. As we achieved quite good results when using compression to fit a bundle into a QR code, we also applied this to fit more bundles into the local storage. Unfortunately, one can only store strings there. Thus, serialization of binary bundles to a JSON string is necessary. While several 100.000 bundles can easily be stored using compression, over 100 MB of RAM are consumed for the temporary string representation. Even worse, the compression of these strings can easily take several minutes for larger numbers of messages as shown in Figure 9. Here, we limited the compression run time to 30 seconds and were able to store 150.000 bundles locally. On disk they only accounted for about 800 KB, so while plenty more bundles could be stored, the long compression time makes this unfeasible. A direct comparison of the uncompressed and compressed variants with around 16.000 bundles already shows the overhead introduced by the compression. While the plain variant only took 71ms the compression already needs 2636ms. So, in general, the uncompressed storage should be preferred, alternatively, compression could be provided by a WebAssembly module, which might deliver a higher compression speed.

2) User Experience: We used our app with various devices in our setup. Here, we will present some observations we made from a users perspective. A brief overview of B-DTN7 in action on an iPad can be seen in Figure 10. The user interface easily adapts from desktop over tablet class devices to

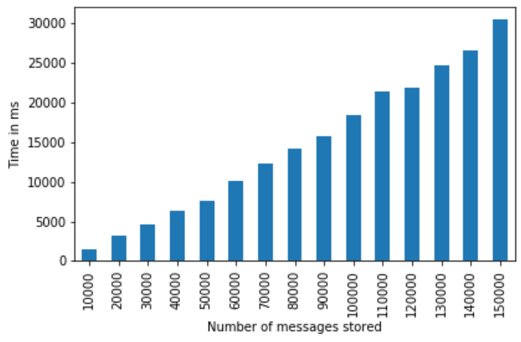
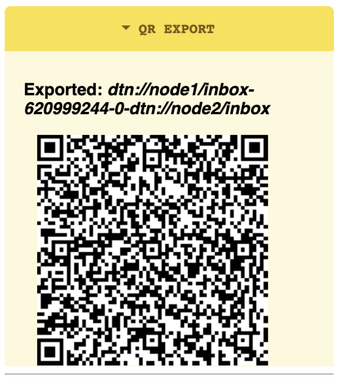


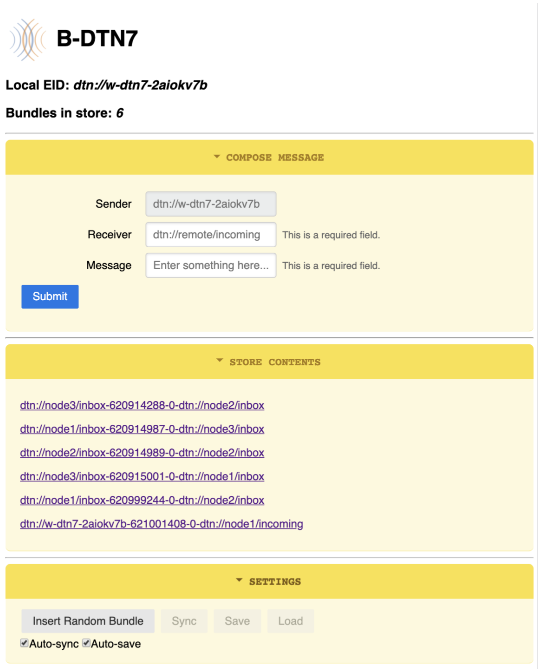
Fig. 9. HTML5 storage speed test with serialized bundles.



QR EXPORT

Exported: *dtn://node1/inbox-620999244-0-dtn://node2/inbox*

Fig. 11. Exported bundle as QR code on iPhone 8



B-DTN7

Local EID: *dtn://w-dtn7-2aiokv7b*

Bundles in store: *6*

COMPOSE MESSAGE

Sender: dtn://w-dtn7-2aiokv7b

Receiver: dtn://remote/incoming — This is a required field.

Message: Enter something here... — This is a required field.

Submit

STORE CONTENTS

dtn://node3/inbox-620914288-0-dtn://node2/inbox
dtn://node1/inbox-620914987-0-dtn://node3/inbox
dtn://node2/inbox-620914989-0-dtn://node2/inbox
dtn://node3/inbox-620915001-0-dtn://node1/inbox
dtn://node1/inbox-620999244-0-dtn://node2/inbox
dtn://w-dtn7-2aiokv7b-621001408-0-dtn://node1/incoming

SETTINGS

Insert Random Bundle | Sync | Save | Load

✓Auto-sync ✓Auto-save

Fig. 10. B-DTN7 web app running on iPad

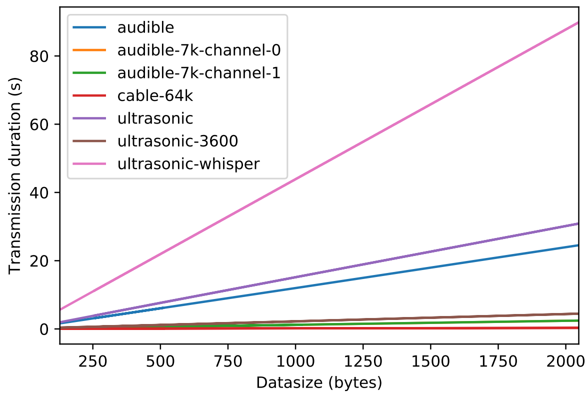


Fig. 12. Transmission duration of bundles with different sizes and different modem profiles.

smartphones. Given a camera is present and web browser with HTML5 support is used, QR codes such as the one generated on an iPhone 8 (Fig. 11), can quickly be transferred.

The biggest limitation from a user perspective is the time it takes to sync a bundle from one device to another even when both are connected to the same access point. While a synchronization is directly triggered when a user presses the send button, the passive sync happens only periodically. This could be compensated by a lower sync interval but this would lead to an increased load and higher battery drain. A better solution would be to use WebSockets for communication or receive push notifications through a service worker and avoid unnecessary synchronizations completely.

Transmitting a bundle via QR code is quite fast but really only feasible for small text payloads. The audio transmission on the other hand can transfer bundles of any size.

In order to benchmark the transmission speeds of audio transfers we generated sound files using the library implemented by the Quiet Modem Project, available open source[7]. The profiles defined by the software were used to generate audio files for different-sized payloads ranging from 128 bytes up to 64 kilobytes. Figure 12 shows the transmission duration times of the different payload sizes up to 2 kilobytes. The profiles *audible-7k* and *cable-64k* are designed for high-quality cable-based connections, and would not work in speaker-microphone transmissions. The profile *cable-64k* reaches the fastest transmission speeds of up to 118.05 kbps. The profile *audible-7k* is designed for full-duplex transmission and uses two different subcarriers, enabling 13.34 kbps in both directions. The profiles designed for speaker-microphone transmissions are named *audible* reaching 1.30 kbps and three *ultrasonic* variants, reaching 0.35 kbps (*ultrasonic-whisper*), 1.04 kbps (*ultrasonic*) and 7.23 kbps (*ultrasonic-3600*). The variants targeting a speaker-microphone combination use the same principal encodings, but are limited to certain frequency bands and thus induce no additional complexity.

Unfortunately, the longer the bundle needs for transmission, the more error-prone the process gets and it also requires significant patience from the users staying close to each other. In our study, we had problems using this functionality across devices. Google's Chrome browser worked the best, Safari and

[7]https://github.com/quiet/quiet

Firefox proved to be problematic, depending on the OS and devices. Another source for possible problems is the audio equipment in the device itself, bad microphones or speakers can prevent successful transmissions easily. In our tests the *ultrasonic* profile worked best, only showing a small amount of failed transmissions where the *audible* profile produces more errors. To summarize, audio-based transmission is a promising approach for peer-to-peer transmission, of bundles in challenging situations. Especially the multicast functionality, sending a bundle to multiple recipients at once, of audio-based transmissions can be a good extension for certain scenarios.

VI. Conclusion and Future Work

In this paper, we have shown that our proposed solution, B-DTN7, enables DTN communication on any platform that has a modern web browser. To achieve this, we designed and implemented a modular version of the Bundle Protocol version 7 draft in the secure systems programming language Rust. Furthermore, we targeted WebAssembly and the web platform to build a progressive web app that can be used to store, carry, forward, and create new bundles. This app provides two novel and distinct ways to transmit bundles even without any network connection by relying either on QR codes or sound for direct device to device transfers. Finally, we designed a bridging component in Google's Go to deliver the web app, provide a rest API for bundle synchronization and interaction with classic *dtn7* daemons using, for example, the MTCP convergence layer. Through various benchmarks, we have shown the efficiency of the bundle protocol library for different standard operations that are needed frequently. Finally, we have presented a small case study with real-world usage of our proposed solution.

In the future, the proof-of-concept web app could be completely implemented as a single wasm app, eliminating the number of involved technologies, making the code less error-prone and easier to maintain. Also, this might lead to more code-reuse across platforms such as desktop apps compared to just the web. Furthermore, sophisticated convergence layers and synchronization mechanisms based upon web technologies such as WebRTC or WebSockets should be considered.

ACKNOWLEDGMENT

This work was supported by the German Federal Ministry of Education and Research (BMBF) as well as by the Hessen State Ministry for Higher Education, Research and the Arts (HMWK) within CRISP. It was also funded by the LOEWE initiative in Hessen, Germany (Natur 4.0 and emergenCITY), and the Deutsche Forschungsgemeinschaft (SFB 1053).

References

[1] S. Burleigh, K. Fall, and E. Birrane, "Bundle Protocol Version 7," Working Draft, IETF Secretariat, Internet-Draft draft-ietf-dtn-bpbis-14, August 2019, http://www.ietf.org/internet-drafts/draft-ietf-dtn-bpbis-14.txt. [Online]. Available: http://www.ietf.org/internet-drafts/draft-ietf-dtn-bpbis-14.txt

[2] S. Burleigh, "Minimal TCP Convergence-Layer Protocol," Working Draft, IETF Secretariat, Internet-Draft draft-ietf-dtn-mtcpcl-01, April 2019, http://www.ietf.org/internet-drafts/draft-ietf-dtn-mtcpcl-01.txt. [Online]. Available: http://www.ietf.org/internet-drafts/draft-ietf-dtn-mtcpcl-01.txt

[3] K. Scott and S. Burleigh, "Bundle Protocol Specification," Internet Requests for Comments, RFC Editor, RFC 5050, November 2007, http://www.rfc-editor.org/rfc/rfc5050.txt. [Online]. Available: http://www.rfc-editor.org/rfc/rfc5050.txt

[4] M. Demmer, E. Brewer, K. Fall, S. Jain, M. Ho, and R. Patra, "Implementing Delay Tolerant Networking," Intel Research Berkeley and University of California, Berkeley, Tech. Rep., 2003.

[5] S. Burleigh, "Interplanetary Overlay Network An Implementation of the DTN Bundle Protocol," JPL, Tech. Rep., 2007.

[6] M. Doering, S. Lahde, J. Morgenroth, and L. Wolf, "IBR-DTN: An Efficient Implementation for Embedded Systems," in *Third ACM Workshop on Challenged Networks*. ACM, 2008, pp. 117–120.

[7] M. Feldmann and F. Walter, "µPCN - A Bundle Protocol Implementation for Microcontrollers," in *2015 Int. Conf. on Wireless Communications & Signal Processing (WCSP)*. IEEE, 2015.

[8] RightMesh, "Terra: Lightweight and Extensible DTN Library," 2018. [Online]. Available: https://github.com/RightMesh/Terra

[9] A. Penning, L. Baumgärtner, J. Höchst, A. Sterz, M. Mezini, and B. Freisleben, "DTN7: An Open-Source Disruption-tolerant Networking Implementation of Bundle Protocol 7," in *Ad-hoc, Mobile, and Wireless Networks - 18th International Conference on Ad Hoc Networks and Wireless, ADHOC-NOW 2019*, Esch-sur-Alzette, Luxemburg, 2019.

[10] P. Gardner-Stephen, "The Serval Project: Practical Wireless Ad-Hoc Mobile Telecommunications," Flinders University, Adelaide, Australia, Tech. Rep., 2011.

[11] A. Dulaunoy, "Forban: A P2P Application for Link-local and Local Area Networks," 2016. [Online]. Available: https://github.com/adulau/Forban

[12] Open Garden, "Firechat," 2019. [Online]. Available: https://www.opengarden.com/firechat/

[13] S. Schildt, J. Morgenroth, W.-B. Pöttner, and L. Wolf, "IBR-DTN: A lightweight, modular and highly portable Bundle Protocol implementation," *Electronic Communications of the EASST*, vol. 37, 2011.

[14] J. Ronan, K. Walsh, and D. Long, "Evaluation of a DTN convergence layer for the AX. 25 network protocol," in *Proceedings of the Second International Workshop on Mobile Opportunistic Networking*. ACM, 2010, pp. 72–78.

[15] I. Daniluk. (2019) TXQR: Animated QR data transfer with Gomobile and Gopherjs. [Online]. Available: https://divan.dev/posts/animatedqr/

[16] ——. (2019) TXQR: Fountain codes and animated QR. [Online]. Available: https://divan.dev/posts/fountaincodes/

[17] B. Armstrong. (2016) Quiet Modem Project. [Online]. Available: https://quiet.github.io/quiet-blog/2016/03/29/quiet.html

[18] U. Paul, M. Nekrasov, and E. Belding, "EmerGence: A Delay Tolerant Web Application for Disaster Relief," in *Proceedings of the 20th International Workshop on Mobile Computing Systems and Applications*. ACM, 2019, pp. 167–167.

[19] K. Sankaran, M. C. Chan, L.-S. Peh *et al.*, "Dynamic framework for building highly-localized mobile web DTN applications," *Computer Communications*, vol. 73, pp. 56–65, 2016.

[20] P. Lieser, F. Alvarez, P. Gardner-Stephen, M. Hollick, and D. Boehnstedt, "Architecture for responsive emergency communications networks," in *2017 IEEE Global Humanitarian Technology Conference (GHTC)*. IEEE, 2017, pp. 1–9.

[21] F. Álvarez, L. Almon, P. Lieser, T. Meuser, Y. Dylla, B. Richerzhagen, M. Hollick, and R. Steinmetz, "Conducting a Large-scale Field Test of a Smartphone-based Communication Network for Emergency Response," *arXiv preprint arXiv:1808.04684*, 2018.

[22] M. Erdelj, M. Król, and E. Natalizio, "Wireless sensor networks and multi-UAV systems for natural disaster management," *Computer Networks*, vol. 124, pp. 72–86, 2017.

[23] P. Graubner, P. Lampe, J. Höchst, L. Baumgärtner, M. Mezini, and B. Freisleben, "Opportunistic named functions in disruption-tolerant emergency networks," in *Proceedings of the 15th ACM International Conference on Computing Frontiers*. ACM, 2018, pp. 129–137.

[24] A. Lindgren, "Social networking in a disconnected network: fbDTN: facebook over DTN," in *Proceedings of the 6th ACM workshop on Challenged networks*. ACM, 2011, pp. 69–70.

[25] L. Baumgärtner, P. Gardner-Stephen, P. Graubner, J. Lakeman, J. Höchst, P. Lampe, N. Schmidt, S. Schulz, A. Sterz, and B. Freisleben, "An Experimental Evaluation of Delay-Tolerant Networking with Serval," in *IEEE Global Humanitarian Technology Conference (GHTC '16)*. IEEE, 2016, pp. 70–79.

[26] C. Bormann and P. Hoffman, "Concise Binary Object Representation (CBOR)," Internet Requests for Comments, RFC Editor, RFC 7049, October 2013.