BLF: A Blockchain Logging Framework for Mining Blockchain Data

Paul Beck[1], Hendrik Bockrath[1], Tom Knoche[1], Mykola Digtiar[1], Tobias Petrich[1], Daniil Romanchenko[1], Richard Hobeck[2], Luise Pufahl[2], Christopher Klinkmüller[3] and Ingo Weber[2]

[1]*Technische Universitaet Berlin, Berlin, Germany*
[2]*Software & Business Engineering, Technische Universitaet Berlin, Berlin, Germany*
[3]*CSIRO Data61, Sydney, Australia*

Abstract
Blockchain technology is increasingly used to realize decentralized applications and execute cross-organizational processes. Understanding how an application is used and how partners and users participate is essential to avoid failures and plan improvements. This understanding can be built by analyzing logs; but although data is in principle given in the immutable ledger, log extraction is currently still inconvenient, slow, and subject to interpretation. In this demo, we present BLF, an extensible logging framework for decentralized applications deployed on a blockchain. The framework is realized for Ethereum and Hyperledger, and has been tested for applications on those networks, but is extensible for other blockchains. Practitioners can use it to analyze their blockchain application and BPM researchers can explore with it new types of event data – event logs from blockchain applications.

Keywords
Logging, Blockchain Application, Process Mining

1. Introduction

Blockchain technology enables a new generation of applications, commonly referred to as *decentralized applications* (DApp), which can e.g., support the execution of cross-organizational business processes [1]. Whereas DApp developers have full control over their DApp's features, the shared nature of the networks on which the DApps are deployed limits the developers influence on when, where, and under what circumstances they are executed. Thus, the analysis of DApp behavior based on logs is essential for avoiding failures and planning improvements for the future. Here, process mining techniques [2] support the analysis of events over time and can provide useful insights as e.g., shown in [3, 4].

In this demo, we present the *Blockchain Logging Framework* (BLF) whose main components are summarized in Fig. 1. At heart, BLF enables the generation of logs from DApp data by allowing users to define an *extract-transform-load* (ETL) pipeline. To this end, users have to

Proceedings of the Demonstration & Resources Track, Best BPM Dissertation Award, and Doctoral Consortium at BPM 2021 co-located with the 19th International Conference on Business Process Management, BPM 2021, Rome, Italy, September 6-10, 2021
⟨firstname⟩.⟨lastname⟩@tu-berlin.de (R. Hobeck); ⟨firstname⟩.⟨lastname⟩@tu-berlin.de (L. Pufahl); christopher.klinkmueller@data61.csiro.au (C. Klinkmüller); ⟨firstname⟩.⟨lastname⟩@tu-berlin.de (I. Weber)

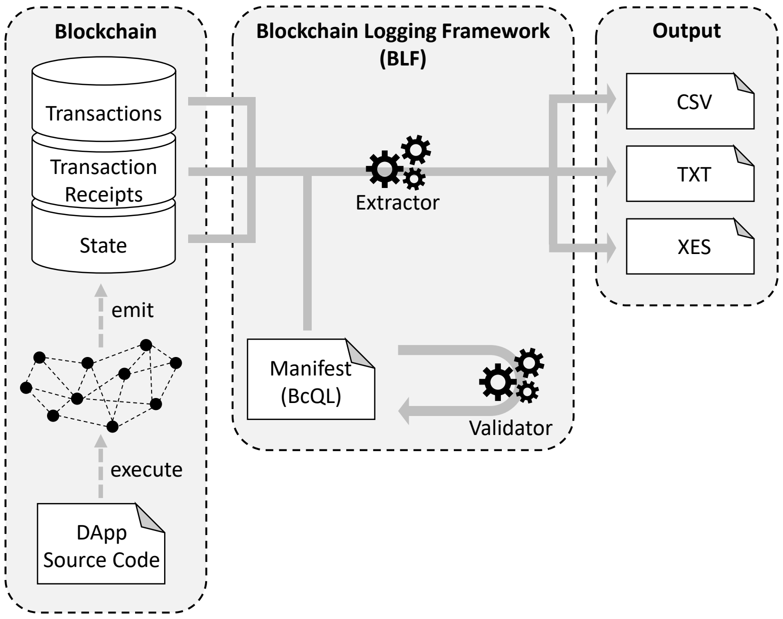

Figure 1: Overview of the blockchain logging framework.

specify a manifest using BLF's *Blockchain Query Language* (BcQL). To support and ease the definition of the manifests, BLF's validator can verify their correctness and inform users about potential specification errors. The framework itself is an extension of the *Ethereum Logging Framework* [4, 5] (ELF) which was developed and tested for Ethereum [6]. BLF by contrast is designed as a generic framework for generating logs from DApps on any blockchain platform. While BLF currently provides adapters for Ethereum and Hyperledger [7], it is extensible so that other platforms can be supported in the future. In the remainder, we present the main functionality, its query language, and the possibility to extend it. We conclude by outlining case studies and a small demonstration for a Hyperledger DApp.

2. Main Functionality of BLF

BLF is written in Java and its source code is publicly available[1]. It consists of three parts: BcQL, the *validator*, and the *extractor*. In the following, we briefly present how BcQL can be applied to specify manifest files and thus ETL pipelines for DApps. After that, we summarize the extractor and validator functionality which builds upon manifest files (see Fig. 1). Lastly, we elaborate on possibilities to extend the framework.

Manifest and BcQL. A manifest file defines how logs are generated from DApp data. This includes details regarding which blockchain to connect to, what data to extract, how to structure the data, and where to store it in which format. To this end, BcQL is designed as a declarative

[1]https://github.com/TU-ADSP/Blockchain-Logging-Framework

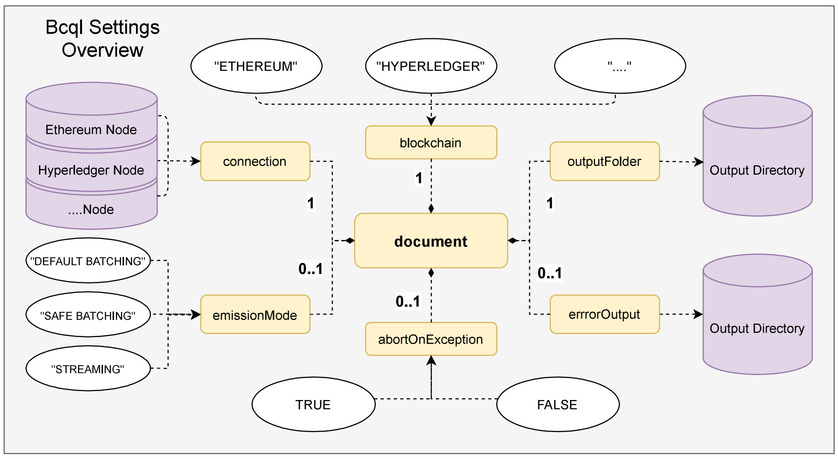

Figure 2: Manifest structure

query language that abstracts away low-level extraction details like data decoding, composition of API calls, etc., so that developers can focus on defining the actual ETL process.

In Fig. 2, the structure of a manifest document is shown with its mandatory and optional elements. In each manifest the user first has to define the blockchain context (e.g. “ETHEREUM”, “HYPERLEDGER”), a connection to a blockchain node, and an output folder. Additionally, BcQL gives the option to specify (1) filters (e.g., block filter, transaction filter), which allow users to narrow down the DApp data to be extracted, (2) expression statements which provide transformation and logic operators that can be used to process the data, and (3) emit statements for formatting data in a specific target format. Additionally, the user may configure the emission mode and the error handling strategy. By default, emission of the output files is done as soon as all data of a blockchain application has been extracted, transformed, and loaded. *Safe batching* and *streaming* allow to emit data for each processed block whereby the former option continuously updates the main output files and the latter produces new files for each block. Regarding error handling, BLF is capable of handling runtime errors in two distinct ways: errors are either ignored, or they lead to the abortion of the current execution. Either way, the errors will get printed to the console and written to an error log file. A detailed step-by-step tutorial on how to write a manifest is provided on the project website[2].

Validator and Extractor. The main components of BLF are the validator and the extractor which process user-defined manifest files (see Fig. 1). Here, we briefly summarize these two components. The interested reader can find more information on these functions in [5].

The *validator* supports the user and checks the manifest for specification errors. In a first step, the parser generator library ANTLR4 *parses* a textual manifest file based on a specification of BcQL’s grammar. In this regard, ANTLR4 generates an intermediate representation of the manifest and identifies syntactic errors. Semantic analysis of the intermediate representation is implemented as a set of custom rules that, e.g., check if filters are correctly nested and that

[2]https://github.com/TU-ADSP/Blockchain-Logging-Framework/wiki/Manifest

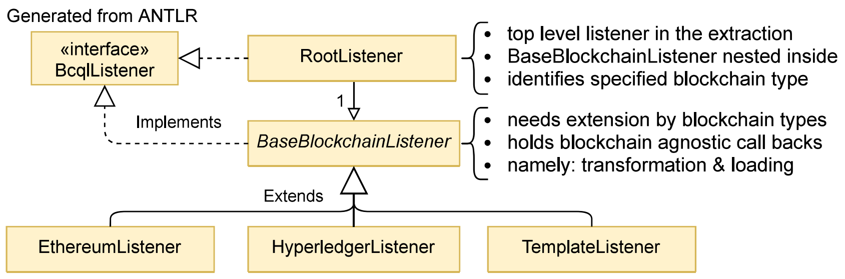
Generated from ANTLR
«interface» BcqlListener
RootListener
top level listener in the extraction
BaseBlockchainListener nested inside
identifies specified blockchain type
1
Implements
BaseBlockchainListener
needs extension by blockchain types
holds blockchain agnostic call backs
namely: transformation & loading
Extends
EthereumListener
HyperledgerListener
TemplateListener

Figure 3: Listeners to Extend BLF to new Blockchain technologies.

variable, parameter, and literal types are compatible.

With a validated manifest, the *extractor* is able to extract, transform, and load data from DApps. The framework extracts data block by block in their historical order, i.e., how they were created and included in a blockchain. During the extraction, the specified filters are considered. For transformation BLF provides a basic set of operators and additionally allows users to integrate custom operators at compile time of the Ethereum Logging Framework. Finally, data can be formatted and exported as textual *application logs*, or in the *comma-separated values* (CSV) and the *eXtensible Event Stream* (XES) [2] format.

Extending BLF. The interaction between BLF and a blockchain node is done through a standardized interface, called *BaseBlockchainListener* (see Fig. 3). By implementing this interface for specific blockchain platforms, e.g., for Corda R3 or EOS, developers can add support for additional platforms. Currently, BLF provides standard implementations for Hyperledger and Ethereum. Besides functionality to extract data from blocks, transactions, log entries, and the blockchain state, developers must declare the default variables that these entities have. For example, on Ethereum blocks are identified by a block number, i.e., the position of the block in the blockchain, while transactions and log entries are identified by indices that encode their position within a block. Developers can follow the step-by-step tutorial[3] to integrate further platforms.

3. Demonstration and Maturity

ELF, the predecessor of BLF, was already used in several case studies, amongst others to examine the popular Ethereum game *CryptoKitties* [4], and the Ethereum DApp *Augur* [3], a popular prediction and betting market. These case studies demonstrate real-world applications of the framework and possibilities how it can be used. After reworking ELF to the extensible BLF, we wrote a BcQL manifest again for *Augur*, ran BLF and successfully demonstrated BLFs continuous capability to extract event logs from Ethereum DApps.

The availability of Hyperledger use cases from production environments is low, because Hyperledger is used as a private permissioned blockchain and has no public blockchain system.

[3]https://github.com/TU-ADSP/Blockchain-Logging-Framework/wiki/Adding-a-new-Blockchain-to-the-BLF

Thus, we implemented our own DApp on a Hyperledger node: *HyperKitties*[4], a Hyperledger reimplementation of the *CryptoKitties* Ethereum smart contract. By porting CryptoKitties to Hyperledger we wanted to test whether the event log generation from Hyperledger results in reasonable event logs as observed in previous case studies. We used *HyperKitties* to write events into our private Hyperledger blockchain. We then created a manifest file that lets BLF connect to the local Hyperledger node, extract the HyperKitties events from the blocks, and generate an event log. We opened it in Disco where we could validate a reasonable event log similar to the Ethereum result. Examples on how to write a manifest and a screencast are provided on the main project website (see Footnote 1).

This demo presented a framework to log data of blockchain-based applications. It provides functions to extract, transform, and load data. The framework additionally supports different modes for data emission and exception handling. It has been already applied in larger case studies and offers BPM researchers a new source of data for process mining. Albeit being usable on production systems, BLF's maturity should still be classified as a fully functional research prototype. In future, we want to extend it to other blockchain technologies and improve the usability of BcQL's grammar.

References

[1] X. Xu, I. Weber, M. Staples, Architecture for blockchain applications, Springer, 2019.

[2] W. Van Der Aalst, Process mining - Data science in action, Springer, 2016.

[3] R. Hobeck, C. Klinkmüller, H. D. Bandara, I. Weber, W. van der Aalst, Process mining on blockchain data: A case study of augur, in: BPM 2021, accepted, Springer, 2021.

[4] C. Klinkmüller, A. Ponomarev, A. B. Tran, I. Weber, W. van der Aalst, Mining blockchain processes: Extracting process mining data from blockchain applications, in: BPM 2019: Blockchain Forum, Springer, 2019, pp. 71–86.

[5] C. Klinkmüller, I. Weber, A. Ponomarev, A. B. Tran, W. van der Aalst, Efficient logging for blockchain applications, arXiv preprint arXiv:2001.10281 (2020).

[6] G. Wood, et al., Ethereum: A secure decentralised generalised transaction ledger, Ethereum project yellow paper 151 (2014) 1–32.

[7] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, et al., Hyperledger fabric: a distributed operating system for permissioned blockchains, in: EuroSys conference, 2018, pp. 1–15.

[4]https://github.com/TU-ADSP/HyperKitties