IPA

2016 年度　未踏 IT 人材発掘・育成事業　採択案件評価書

1. 担当 PM

竹迫 良範 PM（株式会社リクルートマーケティングパートナーズ 専門役員）

2. 採択者氏名

クリエータ：楾本 論（株式会社 PTP）

3. 委託金支払額

885,200 円

4. テーマ名

Web 技術を利用したモダンなパケットアナライザの開発

5. 関連 Web サイト

https://github.com/dripcap/dripcap

6. テーマ概要

本プロジェクトでは、ネットワークデバッグツールの選択肢を増やし、ネットワークプログラミングの開発環境を改善するために、より現代的なコンセプトで新しいパケットアナライザ「Dripcap」を設計・開発した。モダンな Web 技術を利用することで、JavaScript で拡張機能を簡単に開発できるパッケージ管理システムを搭載し、プロトコルの設計やデバッグをより効率的に行うことができる。近年のネットワークアプリケーションプログラミングの環境では、IoT の分野では通信の軽量化や安定性が必要とされ、HTTP/2 などの例に見られるように、アプリケーション層のプロトコルであってもバイナリプロトコルが採用されるようになっており、独自プロトコルを解析できる柔軟なデバッグ環境が必要とされている。現在の OSS のパケットアナライザでは Wireshark が有名であり、数多くのプロトコルに対応しているが、ソフトウェアの設計思想が古く、現在の OS やハードウェア環境を十分に活かすことができておらず、新しい独自プロトコルへの柔軟な対応が課題であった。

7. 採択理由

自作のパケットアナライザ Dripcap の開発提案で、Dripcap は HTML5/CSS/JavaScript の Web 技術で実装することで、独自プロトコルに対応するプラグインも JavaScript で簡単に書くことができる。Canvas や WebGL の技術と組み合わせることによって、時間軸でのパケットのやり取りをアニメーションで可視化できる拡張性も持つ。キャプチャ部分など OS 依存の部分がいくつかあるが、共通基盤の開発を継続することでパケットアナライザの新しいプラットフォームとなることを期待し、採択した。

8. 開発目標

本プロジェクトでは、以下の 6 点を主な開発目標とした。

- Web ベースの柔軟な API とレイアウトシステムの実装
- パフォーマンス改善と高速なパケット処理の実現
- 複数プロトコルの解析結果の提示
- 高度なパッケージ管理システムの提供
- プラグイン開発に便利なライセンス形態
- インストールの簡略化

9. 進捗概要

本プロジェクトでは、Electron をベースにした新しいグラフィカルパケットアナライザを開発した（図 1）。新しく開発した Dripcap の UI のレイアウトシステムでは全部で 4 つの領域に分かれており、それぞれの領域の中央・上・下・右・左に任意のコンポーネントを配置することが可能である。中央の領域には複数のコンポーネントの配置が可能で、それぞれタブで切り替える。この仕組により UI コンポーネントのパッケージを個別に ON・OFF しても全体のレイアウトを崩さずに更新できる（図 2）。

Dripcap では Web ブラウザのレンダリングエンジンをベースとした UI を採用しているので、スクリプトから DOM を自由に操作することができる。HTML/CSS にとどまらず、Canvas API や WebGL などの高性能なグラフックス API に対応している。文字列だけでなく画像や 3D オブジェクトなどを挿入でき、複雑なデータ構造やアプリケーション固有のデータなどを、より視覚的にわかりやすく表現することができる（図 3）。

図 1 開発した Dripcap の画面

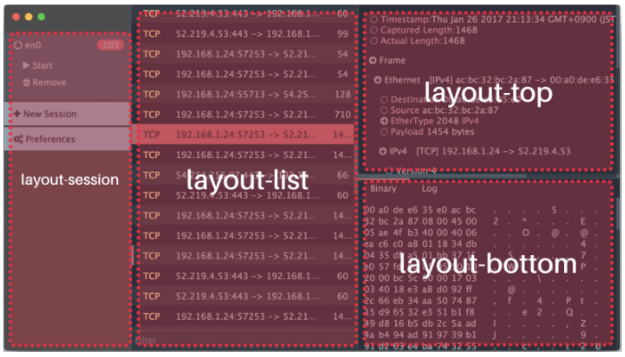

図 2 Dripcap の UI レイアウトシステム

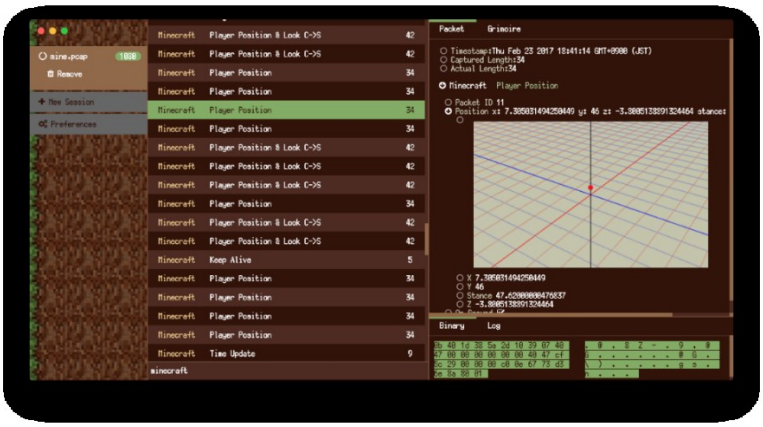

図 3 座標を 3D 空間にプロットする

最初のパケット解析エンジンの設計は、解析用のプロセスを複数作成し、それぞれのプロセスで JavaScript エンジンを独立に実行し、MessagePack を使って解析結果をシリアライズしてプロセス間通信で UI プロセスに転送する方法だった。この方式は開発プロセス間の独立性が高く、安定して動作するという点では優れているが、MessagePack のシリアライズのコストが大きくパフォーマンスが悪化するうえ、複数のプロセスに共通したデータをコピーしなければならないことがあり、メインメモリの消費量が大きくなってしまうという問題があった。改良版では、パケット解析用のプロセスを UI プロセスに統合し、JavaScript のスレッドを複数作成することで並列実行するようにした。パケットのデータはシリアライズされない状態でプロセス内部で共有されているので、データが重複せずアクセスも非常に高速になる。JavaScript の内部ではパケットのデータを C++のスマートポインタへの参照として持っており、データ自体は読み取り専用になっている。C++のスマートポインタでは参照カウントの増減はアトミックに実行されるため、JavaScript の GC が非同期に発生しても問題なくオブジェクトの寿命を管理することができる。また、解析スレッド上でログメッセージが発生したときに、メッセージをキューに追加した後にイベント

ループに対してイベントを通知するように改良した。Node.js 上では次のイベントループが実行されるタイミングでイベントがあるかどうかチェックを行い、メッセージがある場合はキューから取り出し、JavaScript のイベントを発行する。この方式では一度に大量の同じようなメッセージが発生するような状況でも、UI 側のイベントループをブロックせずにログのリダイレクトができるようになる（図 4）。

あるパケットを解析するときに、下位のプロトコルのヘッダの内容で上位のプロトコルの内容をはっきりと判別できる場合は問題ないが、実際には推定が難しいプロトコルも多い。Dripcap ではこのような場合にプロトコルスタックの構造が分岐し、複数の解析パスを並列して実行することで、何通りものプロトコル解析結果を表示できるようになっている。例えば、DNS のパケットとして解析ができていたとしても、ポート番号が 53 でない場合は DNS の可能性は低い。UI 上では、可能性の高いプロトコルほど上に表示され、パケットの一覧では最上位のプロトコルの名前が表示されるが、可能性が低いプロトコルの解析結果も同時に確認できる（図 5）。

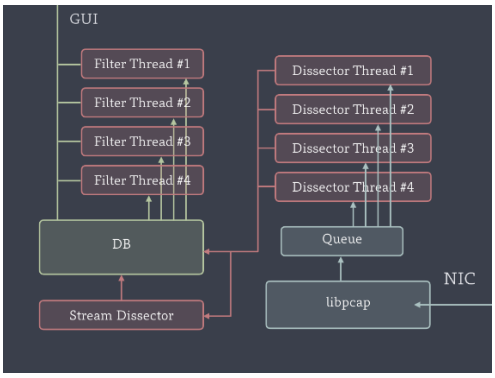

図 4 解析エンジンの処理フロー

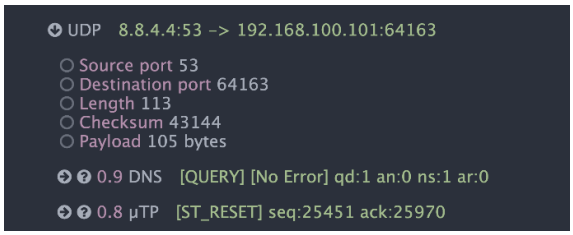

図 5 複数のプロトコル解析結果

Wireshark の実装では、ディスプレイフィルタを実行するときにフィルタの構文木を独自のフラットなバイトコードに変換し、それを内部の VM で実行している。複雑なフィルタの場合、バイトコードの状態で最適化が行えるという利点はあるものの、一般的に VM はオーバーヘッドが大きくパフォーマンスの面で問題がある。Dripcap では VM を構築せずに C++11 のラムダ式を利用し、フィルタの構文木を直接使って関数の合成を行い、C++の関数オブジェクトとして直接実行することでオーバーヘッドを減らしている。マルチスレッドによる並列処理の効果と合わせることで、Wireshark のディスプレイフィルタの 2.5 倍以上のパフォーマンスを実現することができた（図 6）。

配布用パッケージビルドの自動化のために、CI 用のクラウドサービスとして、Linux と macOS 用に TravisCI、Windows 用に AppVeyor を採用し、Git のタグ

を追加するだけで配布用のバイナリパッケージを自動作成して GitHub に自動アップロードするようにした（図 7）。

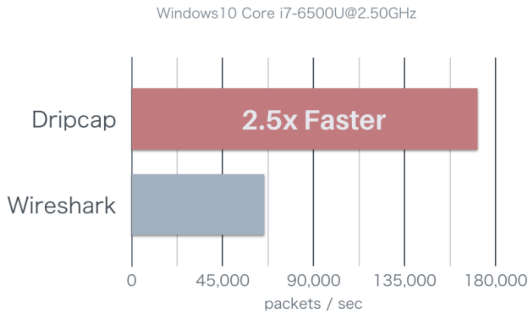

図 6 Wireshark とのパフォーマンス比較

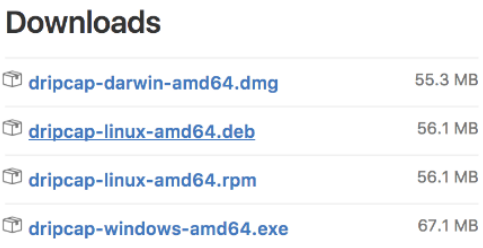

図 7 Github 上で配布しているバイナリ

10. プロジェクト評価

本プロジェクトは、モダンな Web 技術を用いて新しいパケットアナライザを開発することが目的であったが、Electron 自体の技術的な制約や、Node.js との共存問題など、JavaScript エンジンに起因する根本的なアーキテクチャの部分で大きく設計変更を検討するための時間をとられてしまった。実行速度に影響する JavaScript のパフォーマンス改善については他 PM と合同で開催した進捗報告合宿でも議題になり、ベンチマークとなる既存の Wireshark のソースコードも一晩で読み込んでみるという研究課題も PM から課した。その結果、Wireshark はシングルスレッドで動いており、CPU のマルチコアの性能を引き出せていないことがわかり、Dripcap は複数スレッドを並列に協調して動作させることでパフォーマンス改善を試みることにした。そして、フルスクラッチで何度もコードを書き直すことで、最善のアーキテクチャを手探りな状況の中で探求し続けて、パケットを処理する JavaScript の実行パフォーマンスを最終的に大きく改善することができたのは大いに評価したい。

11. 今後の課題

Dripcap は UI デザインも採用技術もモダンで、非常に完成度の高いプロダクトに仕上がったが、今後は幅広いプロトコルに対応していくために、いかにオープンスースコミュニティを巻き込んで拡張プラグインを充実させていくかが大きな課題と考える。プロジェクト期間中は、アーキテクチャの大幅な変更を何度も予定していたため、プラグインの利用する関数はすべて内部用 API で後方互換性をあえて保障しなかったことも少なからず影響していたと思われるが、自分一人で対応プロトコルを増やしていくには時間的な制約もあってスケールさせるには限界がある。今後はブランチを区切ったリリースマネージメントも実施しながらも、プラグイン機構の API 後方互換性を一定レベルで保ち、外部

のプラグイン提供者をいかに増やし、どのようにして段階的にエコシステムを構築していくか課題である。