Unobtrusive Mechanism Interception

Patrick Lampe, Markus Sommer, Artur Sterz, Jonas Höchst, Christian Uhl, Bernd Freisleben
Department of Mathematics & Computer Science, University of Marburg, Germany
{lampep, msommer, sterz, hoechst, uhlc, freisleb}@informatik.uni-marburg.de

***Abstract*—Networked systems and applications are often based on proprietary hardware/software components that manufacturers might not be willing to adapt or update if new requirements arise. We present *mechanism interception*, a novel approach to unobtrusively add or modify functionality to/of an existing networked system or application without touching any proprietary components. Behavioral changes are achieved by functionality-enhancing yet unobtrusive interceptors, i.e., components introduced between systems and their environments adding or updating mechanisms. We illustrate our approach by unobtrusively adding a vertical handover mechanism between Wi-Fi and LTE to a mobile end device without disconnecting TCP sessions. Our results indicate that mechanism interception is a compelling approach to achieve improved service quality and provide previously unavailable functionality.**

I. Introduction

Networked systems are often based on proprietary hardware/software components (e.g., applications, access points) and communication between them (e.g., network interfaces, system calls) that manufacturers might not be willing to adapt or update if new requirements arise. Possible reasons are: insufficient monetization of deprecated components and/or technical hurdles such as missing infrastructure. Furthermore, long and costly standardization and deployment processes may also hinder or slow down updates, which is not only true for proprietary components. Generally, this leaves users with no choice but to accept the non-optimal functionality of networked systems.

To add or modify functionality to/of existing networked systems or applications without touching proprietary components, developers use abstractions and workarounds that unobtrusively intercept and modify the communication between proprietary components and their environments. For example, developers introduced Network Address Translation (NAT) as a solution for insufficient IPv4 addresses, or Wine to run Windows applications on Unix-like operating systems.

We argue that such solutions are not exceptions, but essential for networked systems to provide highly desired improvements in a fast manner. However, there is no approach that can be used by developers to systematically identify possible starting points and implement such functionality.

Therefore, we present *mechanism interception*, a novel approach to implement functional additions to or modifications of existing networked systems without touching any proprietary components. We distinguish between *system*, i.e., components containing non-changeable parts, and *environment*, i.e., components containing modifiable parts under control. Behavioral changes are achieved by functionality-enhancing yet unobtrusive *interceptors*, i.e., hardware/software components that are introduced between environment and system to add or update mechanisms representing some functionality. Examples of interceptors are an updated software library, a newly deployed edge device, or an enhanced cloud service. Interceptors must be unobtrusive to avoid disrupting or even destroying applications, but still provide added or modified mechanisms to the networked systems. We illustrate our approach by a case study, where we unobtrusively add a vertical handover mechanism between Wi-Fi and LTE to a mobile device without disconnecting existing TCP sessions. Our results indicate that mechanism interception is a compelling approach to achieve improved service quality and provide previously unavailable functionality. in an unobtrusive manner.

II. Related Work

The basic idea of making network protocols extensible has been investigated in the context of active networks [1], [2]. ANTS [3] enables new protocols to be deployed on routers as well as terminals through platform-independent code. Pathak et al. [4] propose a system that provides applications with additional TCP interfaces and makes TCP adaptable in a fine-grained way. With software-defined networks [5] and the network programming language P4 [6], network programmability is provided. Tran et al. [7] present a method to add functionality to the TCP kernel implementation using eBPF. De Coninck et al. [8], [9] present a method to dynamically tune QUIC for each connection via extensions.

Alpine [10] and MultiStack [11] are userspace implementations of network stacks that allow changes to be tested quickly. An implementation of TCP in userspace is presented by Jeong et al. [12]. With NUSE, the network stack of the Linux kernel can be used as a userspace library [13] and existing programs can use modified protocols without modifications to the program itself. Heuschkel et al. [14] present VirtualStack, which allows different userspace network stacks to be used in one system. ClickNF [15] is an extension of a software-defined router in which the lower four layers of the network stack can be exchanged in a modular way.

Balinsky et al. [16] describe a method to prevent data leaks with system call interception and DAVINCI [17] uses system call hooking to build a fully transparent dynamic analysis tool for Android apps. Somy et al. [18] propose a sandbox architecture for serverless functions based on system call monitoring and whitelisting.

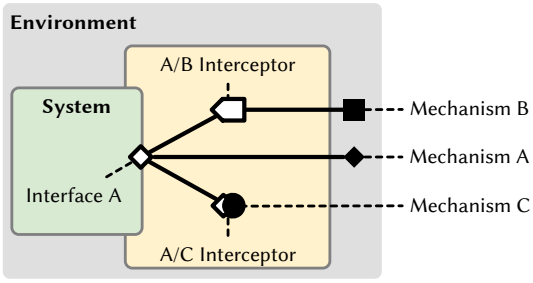

Fig. 1. System, Environment, and Interceptor

III. Unobtrusive Mechanism Interception

We motivate the novel concept of unobtrusive mechanism interception by the following examples:

1) Network Address Translation: The Internet Protocol version 4 (IPv4) was designed in the late 1970s, allowing only roughly 4.3 billion hosts in the Internet. Although there is a newer version, IPv6, for quite some time, allowing 2^{128} hosts in the Internet, its roll-out has been very slow. To tackle the looming threat of address scarcity, Network Address Translation (NAT) has been introduced. Using NAT, it is not necessary to update or somehow alter the hosts of the private network, since NAT unobtrusively translates addresses without private hosts even noticing it.

2) Wine: To allow Windows applications to be executed on Linux or other POSIX-compatible operating systems, the Wine project[1] was initiated. It re-implements Windows system calls and libraries, such that system calls of Windows applications are translated to their corresponding POSIX equivalents.

A. Mechanism Interception

In both examples, there is a system (i.e., a private host or a Windows application) that cannot be upgraded or modified. It uses mechanisms (i.e., IP routing or system calls) to achieve a specific task (i.e., communications between networks or executing a Windows application). Developers can control the environment (i.e., the NAT router or the operating system). and use a new mechanism that intercepts information from an old mechanism and translates it to the new mechanism (i.e., translating between private and public IP addresses or translating between Windows and POSIX system calls).

Our novel approach of unobtrusive mechanism interception is shown in Fig. 1. It consists of the following components:

1) System and Environment: We consider a system (green in Fig. 1) that is intended to achieve a particular task. The system or parts of it cannot be changed because it contains proprietary components or depends on other external factors, such as high coordination costs for changes due to long standardization or technical hurdles. In the NAT example, the system is the host that wants to communicate with hosts in other networks. The system is surrounded by an environment (gray in Fig. 1) that provides functionality to be used by the system. The system depends on the functionality provided by the environment to achieve the intended task. In the NAT example, the host needs the routing functionality of the gateway to enable communication with hosts in other networks.

2) Mechanism: A mechanism is the implementation of a functional unit of the environment used by the system via its corresponding interface to achieve its task [19]. The mechansim/interface mapping is not unique. For example, for the TCP mechanism, the interface could be the Sockets API or the Wi-Fi connection between two devices. In Fig. 1, the mechanisms are represented in black geometric figures and their interfaces in corresponding white geometric figures. In the NAT example, the mechanism provided by the environment that the system uses is IP routing and the corresponding network interface between host and NAT router.

3) Unobtrusive Interceptor: In the area of software engineering, the "interceptor architectural pattern allows services to be added transparently to a framework and triggered automatically when certain events occur" [20]. In this programming pattern, a framework provides interfaces so that programs using the framework can transparently intercept the flow of data at specific events. While the goal of our interceptor is similar, i.e., to transparently intercept the data flow to enable new functionality, the perspective is reversed. An interceptor in our approach is part of the environment and provides mechanisms to the system via interfaces, which are represented as combinations of geometric figures in the yellow area in Fig. 1. In contrast to the interceptor pattern, our interceptor is used by the system but configured by the environment. To allow a system to use the mechanisms, the interceptor is inserted between the system and the mechanism and changes the data flow. The used mechanism can also be introduced into the environment as part of the interceptor and does not have to exist before. In the NAT example, the newly added mechanism is *IP masquerading*.

To not influence or even disturb the actual task of the system, the existing mechanism must be unobtrusively replaced by the provided mechanism of the interceptor. The system should not notice the change of the mechanism; the used interfaces pretend that the original mechanism is still in place. Furthermore, the behavior of an interceptor must be unobtrusive so that the system can continue to provide the functionality and not provide error cases for its termination.

B. Guide to Action

To implement unobtrusive mechanism interceptors, we propose a 3-step procedure for developers.

1) Identification of Problem and Solution: The first step is to identify the problem that needs to be solved. Usually, problem identification is the result of prolonged use of a system where limitations occur or are noticed. Once the problem is identified, a solution must be found. Typically, there are several possibilities from which feasible solutions can be distilled. In the NAT example, the problem is IPv4 address scarcity, and the solution might be to translate addresses between multiple networks or to adjust routing on hosts so that unique IPv4 addresses are not necessary.

[1]https://www.winehq.org

2) Identification of System and Environment: In the second step, developers need to identify the system in terms the problem to be solved, i.e., the component that cannot be changed or controlled, and the environment that can be controlled or changed by an interceptor. Some solutions of the first step must be discarded, since they would require a change of the system. In the NAT example, the system is the host that communicates with hosts in other networks, and the environment is the local network or NAT router. The alternative suggestion of adapting hosts so that IPv4 addresses do not have to be unique is omitted because the host is not changeable.

3) Identification of Mechanisms and Interceptors: In the third step, mechanisms must be found to solve the problem by considering the feasibility and unobtrusiveness of an interceptor that translates between the old and the new mechanism. This makes it essential that the mechanisms are functionally equivalent, for example, that they are on the same layer in the case of network protocols. In the NAT example, the functionally equivalent mechanism is an adaptation of IP routing. This interceptor implements an alternative IP routing mechanism, i.e., functionally equivalent to the old mechanism. Finally, the interceptor has to be implemented and deployed.

IV. TunnelHandovers

We present the use case of an unobtrusive mechanism interceptor that uses common Linux mechanisms to enable handovers without TCP session disconnects, and without the need to roll out new, complicated, or niche protocols.

A. Problem and Solution

A vertical handover (henceforth simply called "handover") is the process of switching a networked device from one network access technology to another [21]. Probably the best-known example is the switch from a mobile phone's Wi-Fi connection to a cellular connection, e.g., when leaving home on the way to the office. The core problem with handovers are protocols such as TCP, which were designed before wireless communication became popular. Thus, disconnections on the transport layer were not really considered during protocol design. If the connection on the transport layer is interrupted, the connections on the application layer must also be re-established, which can lead to different implications for different applications, such as interruptions of audio streams when listening to music or making phone calls.

B. System and Environment

The system in this scenario is an TCP-based application that we cannot change, e.g., by adding UDP functionality, since it is a pre-compiled application from some app store. The environment is the operating system, which we can change by implementing and deploying an interceptor.

C. Mechanism and Interceptor

We use two building blocks: WireGuard and `LD_PRELOAD`. WireGuard is a simple, fast, and secure VPN software that transmits data encapsulated in UDP datagrams. The use of UDP eliminates the problem of TCP connection losses during handovers, since UDP is not connection-oriented. For a WireGuard tunnel to work, a tunnel endpoint somewhere in the network is required, where the WireGuard tunnel is terminated and the encapsulated TCP connection is further relayed to the original destination.

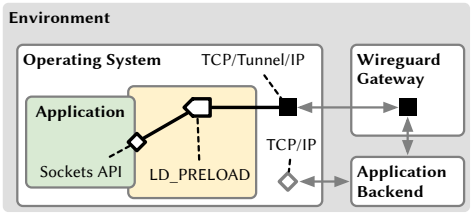

Fig. 2. The system architecture for the TunnelHandover approach

The use of WireGuard alone does not automatically enable existing TCP-based applications to support handovers. Rather, they have to be instructed to transfer their data over the WireGuard connection. The `LD_PRELOAD` mechanism allows intercepting functions of dynamically linked libraries, enabling us to intercept the Socket API such that only connections of the application started with our `LD_PRELOAD` modifications use the WireGuard tunnel. Thus, the mechanism *TCP/IP* is replaced by the mechanism *TCP/Tunnel/IP*. For this purpose, an interceptor is used that implements the Socket API used by the system, i.e., the application.

Fig. 2 presents the components of our TunnelHandover approach. A WireGuard daemon is executed on the operating system, indicated by the black square. When a program is executed with our interceptor, the TCP packets of this particular application are intercepted and redirected to the local WireGuard daemon using `LD_PRELOAD`. The UDP-based WireGuard packets are then sent to the WireGuard endpoint located in the network, which unpacks the encapsulated TCP connection and acts as a relay to forward the application data to the desired destination using conventional TCP/IP mechanisms. The responses of the server with which the application communicates are sent back to the host via the same route, i.e., via the WireGuard tunnel.

D. Experimental Evaluation

We evaluated the proposed interceptor for seamless handovers by emulation using the Common Open Research Emulator (CORE) in a leaving home scenario. Here, the handover is performed from Wi-Fi to LTE. The following nodes were emulated: client (with the TCP-based application to be intercepted), Wi-Fi access point (AP) and LTE access network (AN) to handover between, backbone router where both AP and AN are connected to for the network uplink, WireGuard gateway to terminate the WireGuard connection, and application server running the server of the client's application. To simulate a TCP-based application, we used iPerf3[2], where the client initiates the connection to the server via Wi-Fi at

[2]https://iperf.fr

experiment start. We also conducted experiments without the *TCP/Tunnel/IP* mechanism for comparison, using the regular *TCP/IP* mechanisms of the Linux kernel.

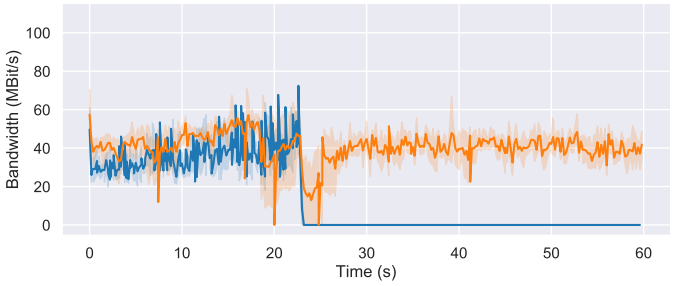

Fig. 3. Network throughput with and without vertical handover

Fig. 3 shows the results for the conducted experiment. The x-axis denotes the experimental time, while the y-axis denotes the achieved bandwidth. The orange graph shows experiments where the proposed handover mechanism is enabled, the blue graph shows the results without employing the *TCP/Tunnel/IP* mechanism. Without the*TCP/Tunnel/IP* mechanism, the throughput drops to 0 because the TCP connection is not re-established after the connection is changed. Using the proposed `LD_PRELOAD` interceptor, a short throughput degradation is visible at around 25 seconds. This is due to the Wi-Fi connection being lost immediately, resulting in a short connection drop until the kernel recognizes the lost connection and propagates the changes to the routing tables.

V. Conclusion

We presented mechanism interception, a novel approach to implement unobtrusive functional additions to or modifications of an existing networked system without touching any proprietary components. Behavioral changes are achieved by mechanism-enhancing yet unobtrusive interceptors. We illustrated our approach by a case study.

There are several areas of future work. Currently, our approach requires manual work to identify system, environment, mechanisms, and interceptors. Providing adequate tools to reduce manual work would help to support the adoption of our approach. Furthermore, apart from adding a single mechanism as part of an interceptor to enhance a single proprietary component, future work should consider supporting multiple mechanisms per system and multiple systems per interceptor. Finally, when an interceptor replaces or modifies an existing mechanism or adds a new mechanism, the old mechanism is still available, although not used. Transitioning between multiple mechanisms could add the benefit of using the mechanism that achieves the best results for a given situation.

Acknowledgments

This work is funded by the Hessian State Ministry for Higher Education, Research and the Arts (HMWK) (LOEWE Natur 4.0, and LOEWE emergenCITY) and the German Research Foundation (DFG, Project 210487104 - Collaborative Research Center SFB 1053 MAKI).

References

[1] D. L. Tennenhouse and D. J. Wetherall, "Towards an Active Network Architecture," *ACM SIGCOMM Computer Communication Review*, vol. 26, no. 2, pp. 5–17, 1996.

[2] D. L. Tennenhouse, J. M. Smith, W. D. Sincoskie, D. J. Wetherall, and G. J. Minden, "A Survey of Active Network Research," *IEEE Communications Magazine*, vol. 35, no. 1, pp. 80–86, 1997.

[3] D. J. Wetherall, J. Guttag, D. L. Tennenhouse, *et al.*, "ANTS: A Toolkit for Building and Dynamically Deploying Network Protocols," in *IEEE Open Archichtecture and Network Programming*, vol. 98, pp. 117–129, San Francisco, CA, IEEE, 1998.

[4] S. Pathak and V. S. Pai, "ModNet: A Modular Approach to Network Stack Extension," in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pp. 425–438, 2015.

[5] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling Innovation in Campus Networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.

[6] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, *et al.*, "P4: Programming Protocol-independent Packet Processors," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, 2014.

[7] V.-H. Tran and O. Bonaventure, "Beyond Socket Options: Making the Linux TCP Stack Truly Extensible," in *2019 IFIP Networking Conference (IFIP Networking)*, pp. 1–9, IFIP, 2019.

[8] Q. De Coninck, F. Michel, M. Piraux, F. Rochet, T. Given-Wilson, A. Legay, O. Pereira, and O. Bonaventure, "Pluginizing QUIC," in *ACM Interest Group on Data Communication*, pp. 59–74, ACM, 2019.

[9] Q. De Coninck and O. Bonaventure, "Tuning Multipath TCP for Interactive Applications on Smartphones," in *2018 IFIP Networking Conference (IFIP Networking) and Workshops*, pp. 1–9, IEEE, 2018.

[10] D. Ely, S. Savage, and D. Wetherall, "Alpine: A User-Level Infrastructure for Network Protocol Development," in *3rd USENIX Symp. on Internet Technologies and Systems*, vol. 3, pp. 15–23, 2001.

[11] M. Honda, F. Huici, C. Raiciu, J. Araujo, and L. Rizzo, "Rekindling Network Protocol Innovation with User-level Stacks," *ACM SIGCOMM Comp. Comm. Review*, vol. 44, no. 2, pp. 52–58, 2014.

[12] E. Jeong, S. Wood, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park, "mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems," in *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pp. 489–502, 2014.

[13] H. Tazaki, R. Nakamura, and Y. Sekiya, "Library Operating System with Mainline Linux Network Stack," *Proceedings of NetDev*, 2015.

[14] J. Heuschkel, A. Frömmgen, J. Crowcroft, and M. Mühlhäuser, "VirtualStack: Adaptive Multipath Support through Protocol Stack Virtualization," in *10th Int. Network Conference (INC)*, pp. 73–78, 2016.

[15] M. Gallo and R. Laufer, "ClickNF: A Modular Stack for Custom Network Functions," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pp. 745–757, 2018.

[16] H. Balinsky, D. S. Perez, and S. J. Simske, "System Call Interception Framework for Data Leak Prevention," in *2011 IEEE 15th Int. Enterprise Distributed Object Computing Conference*, pp. 139–148, 2011.

[17] A. Druffel and K. Heid, "Davinci: Android App Analysis Beyond Frida via Dynamic System Call Instrumentation," in *Int. Conference on Applied Cryptography and Network Security*, pp. 473–489, Springer, 2020.

[18] N. Somy, A. Mondal, B. Ghosh, and S. Chakraborty, "System Call Interception for Serverless Isolation," in *Proceedings of the SIGCOMM'20 Poster and Demo Sessions*, pp. 57–59, New York, NY, USA: Association for Computing Machinery, 2020.

[19] B. Alt, M. Weckesser, C. Becker, M. Hollick, S. Kar, A. Klein, R. Klose, R. Kluge, H. Koeppl, B. Koldehofe, *et al.*, "Transitions: A Protocol-independent View of the Future Internet," *Proceedings of the IEEE*, vol. 107, no. 4, pp. 835–846, 2019.

[20] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, *Pattern-oriented Software Architecture, Patterns for Concurrent and Networked Objects*. John Wiley & Sons, 2013.

[21] J. Höchst, A. Sterz, A. Frömmgen, D. Stohr, R. Steinmetz, and B. Freisleben, "Learning Wi-Fi Connection Loss Predictions for Seamless Vertical Handovers Using Multipath TCP," in *2019 IEEE 44th Conference on Local Computer Networks (LCN)*, pp. 18–25, IEEE, 2019.