On UDP continuity over vertical handovers

Jihun Ha, Jiung Yu, Byungjo Kim, Hyogon Kim

Korea University, 309 Woo Jung Information & Communications Building, Anam-Dong, Sungbuk-Gu, Seoul, Republic of Korea

Abstract

Today’s mobile devices such as smart phones are equipped with multiple wireless access technologies, so multi-radio operation is becoming the norm. These devices frequently face vertical handover scenarios such as exiting Wi-Fi hotspot towards cellular network coverage. Unfortunately, the vertical handover execution is still done in a very primitive way even with middleware support, and the reality is that any active service at the time of handover is aborted or at least severely disrupted when it happens. In this paper, we explore how the transport layer can independently overcome the disruption caused by the vertical handover and obtain service continuity with or without underlying vertical mobility support infrastructures such as Media Independent Handover (MIH) or Access Network Discovery and Selection Function (ANDSF). In particular, we focus on the case of User Datagram Protocol (UDP), and explore how to evolve it for vertical handover. In particular, we show that UDP can be enriched with options while keeping application transparency and enabling incremental deployment as is the case for Transmission Control Protocol (TCP). We implement the proof-of-concept prototype and demonstrate that it performs well under dynamic Cellular-WiFi vertical handover settings with off-the-shelf smart phones operating across commercial 3G cellular carriers and Internet Service Provider (ISP) networks. We also consider the ways to utilize the evolved UDP for other applications, as the addition of carrying options or in-band signaling capability opens a whole new horizon for the usability of UDP.

1. Introduction

Today’s mobile devices enjoy multiple modes of network access. Smart phones, tablet devices and laptops typically have Bluetooth, Wi-Fi, and 3G/4G interfaces. Multi-radio operation is becoming the norm, and users increasingly face vertical handover situations. For instance, smart phones are frequently configured to associate with a wireless local area network (LAN) when it is in a Wi-Fi hotspot, but to revert to cellular access when exiting. Unfortunately, the vertical handover on these smart devices is still executed in a primitive way, and the reality is that any active service at the time of handover is aborted or at least visibly disrupted even with middleware support.

A mid-call vertical handover can be especially painful for the receiving side of User Datagram Protocol (UDP)-based services. This is in contrast to the sending side, where the interface changes are instantly reflected on routing and service continues over the new interface. But if the vertical mobility takes place at the receiving end, the UDP flow gets to be broken since the sending side UDP is never notified of the receiving interface change. For example in Fig. 1, $f_{0}$ is an ongoing UDP flow, during which the UDP receiver vertically moves (i.e., changes its wireless interface) from the Wi-Fi to the cellular connection, hence redefining itself as a modified flow $f_{N}$ with a changed receiver endpoint. But in reality, as the UDP sender is unaware of the change, it continues to send to the old endpoint at the receiver, which is now disabled. Therefore, the dynamic flow mobility scenario as depicted in the
The figure is difficult to support unless there is an in-network mechanism working on behalf of the UDP sender to detect and cope with such change (possibly over multiple technical and administrative domains), or alternatively, there is a novel mechanism that seamlessly involves the UDP sender.

No protocol mechanism for service continuity currently exists to cope with an interface associated with a UDP socket rendered unusable on a vertical handover. The current practice is to rely on the application (or user) to create a new association and resume the application stream from the failed point if not restart. On Android, for instance, the signal from the kernel on the change of the interface status can trigger a middleware (“ConnectivityManager” [1]) to prompt the user to create a new socket for the application on an alternate interface. Not only this approach is non-transparent to the application and/or the user, but any context information (e.g., current position in the flow or codec information, etc.) is lost so it should be freshly provided.

Traditionally, mobility standards and mechanisms have focused on layer 2 (L2) [2–4], layer 3 (L3) [5], or layer (L5) [6], instead of layer 4 (L4). This approach is retained in handling vertical handovers. In particular, the latest standards such as the Institute of Electrical and Electronics Engineers (IEEE) 802.21 Media Independent Handover (MIH) [7] and the 3rd Generation Partnership Project (3GPP) Access Network Discovery and Selection Function (ANDSF) [8] are explicitly designed to work under existing standards such as Mobile IP (MIP) on L3 or Session Initiation Protocol (SIP) mobility [6] on L5. A notable exception is Stream Control Transmission Protocol (SCTP), which has fail-over capability on L4 against dynamic interface changes [9]. Recently, the Internet Engineering Task Force (IETF) has begun to work on multi-path Transmission Control Protocol (TCP) [10]. But interestingly enough, UDP has been so far largely left intact as a passive element in handover frameworks.

Unfortunately, the existing mobility management standards and mechanisms seem to fall short of solving the UDP vertical handover problem. In particular, MIP or MIP variants (including the proposal aiming at vertical handover support [5]) are not adopted universally enough. As for the SIP-based mobility [6], not all UDP-based applications use SIP signaling. MIH and ANDSF are not independent solutions, but assistants to the above mechanisms. Therefore, in the general light of the robustness principle, we need to equip UDP itself to independently deal with interface changes due to vertical handovers and to survive them with or without the support of extrinsic mobility management mechanisms. In doing so, we have the following design principles:

- Extend UDP transparently to applications so that no application programming interface (API) change is necessary.
- Make it incrementally deployable, i.e., backward compatible with legacy UDP peers.
- Make it work across administrative/carrier boundaries and across middleboxes.
- Make it work with existing vertical handover support infrastructure, but work independently if no such support is given.

Note that we take the minimalist approach to the problem, i.e., we do not require additional infrastructure as in MIP variants or in SIP-based mobility. Below, we discuss the enhanced UDP design in detail. We also implement a proof-of-concept prototype on Android smart phones and test it in commercial ISP/3G carrier networks. A slight extension of the UDP vertical handover is UDP multi-path transport. Using XUDP’s capability to deal with dynamic addition and deletion of the receiving interface, we can let a UDP flow to use multiple receiving interfaces. We also discuss the API that is transparent to the legacy UDP users but has enhanced features that work with the new UDP. We leave other important issues such as maintaining security associations and translating Quality of Service (QoS) across, as well as integrating the enhanced UDP features with user preference and policy to future work, as they are beyond the scope of this paper.

2. Related work

In this section, we briefly discuss related work that may be used to deal with the vertical handover of the UDP receiver. For convenience, we classify these work along the protocol stack.

2.1. Bonding/logical interface

In some operating systems, so-called the bonding interface is allowed [3]. The bonding interface abstracts two or more physical interfaces into a single logical interface so that IP can deal with a single abstract interface while the corresponding physical interfaces are used on availability basis. For instance, a server can transition to a backup interface when the primarily used interface fails. The bonding interface renders such transition transparent to the IP layer and up, masking the physical interface disruptions. But as such, this technique applies to a static situation where the physical interfaces are pre-configured. In the mobile context, the acquisition and the loss of physical interfaces are dynamic and unpredictable. More importantly, the physical interfaces under the bonding interface should be on the same subnet so that a single IP address...
assignment on the bonding interface can work. If the physical interfaces are attached to different networks, assigning a single IP address is complicated if not impossible (which is the case when moving between administrative domains). A derived notion from the bonding interface is the Logical Interface [4]. In a Proxy Mobile IPv6 (PMIPv6) domain, the proposed semantics can be useful for vertical handover, multihoming, and flow mobility, if PMIPv6 is extended so that the network elements such as Mobile Access Gateway (MAG) and Local Mobility Anchor (LMA) can deal with the logical interface [5]. Even so, the different interfaces under the same Logical Interface should be in the same administrative domain.

2.2. 3GPP EPC

In the 3GPP Evolved Packet Core (EPC) architecture [2], the mobile can move between different cellular technologies such as GSM EDGE Radio Access Network (GERAN), Universal Terrestrial Radio Access Network (UTRAN), and Evolved UMTS Terrestrial Radio Access Network (E-UTRAN). The vertical handover is transparent to the IP layer, but the type of vertical handover supported by the EPC is restricted to the same administrative (e.g., provider) and technology (i.e., cellular network) domain.

2.3. MIH and ANDSF

The Media Independent Handover (MIH) defined by the IEEE 802.21 standard [7] and the 3GPP Access Network Discovery and Selection Function (ANDSF) [8] are newer standards that are directly related with vertical handover. MIH provides the standardized frameworks for detecting link state change, finding usable access networks in the neighborhood, and controlling the link configurations both locally and remotely. In ANDSF, a user equipment (UE) can be controlled by the 3GPP cellular operator to discover non-3GPP access networks such as Wi-Fi or Worldwide Interoperability for Microwave Access (WiMAX) that can be used for data communications in addition to 3GPP access networks (such as High Speed Packet Access (HSPA) or Long-Term Evolution (LTE)) and to provide the UE with rules policing the connection to these networks. Note that these standards are explicitly designed to assist higher layer mobility management protocols such as MIP or SIP mobility, but not to execute the vertical handover themselves. Both MIH and ANDSF also require the additional infrastructure, e.g., Dynamic Host Configuration Protocol (DHCP) or Domain Name System (DNS) servers for their operations. Also, they can assist only the vertical handover types that occur in the same administrative domain.

2.4. PMIP flow mobility

In the IETF, the PMIPv6 is being extended to cover the flow mobility [5]. It assumes so called the Logical Interface [4] that is similar to the bonding interface discussed above. Once the logical interface abstraction is provided, the extension enables the PMIPv6 to move selected flows from one access technology to another. In general, the PMIP solution is limited within the administrative boundary since it is a network-controlled handover model, as well as it requires the infrastructure support such as MAG and LMA.

2.5. SCTP

On the transport layer, Stream Control Transport Protocol (SCTP) [11] has flexibility to accommodate changes in the endpoint addresses. Designed for high availability environments such as Signaling System 7 (SS7), the original specification allows for multiple paths in one association between a pair of endpoints [11]. Moreover, with its Dynamic Address Reconfiguration (DAR) extension [9], it can deal with dynamic endpoint address acquisition or loss. Although the unordered mode in SCTP is similar to UDP delivery, SCTP cannot replace UDP since it is a reliable transport protocol.

2.6. SIP-based vertical handover

Wu et al. [12] discusses SIP-based vertical handover especially in terms of its delay performance. In this scheme, the mobile registers its new IP address with the redirect or SIP server using the SIP REGISTER message for pre-call mobility. In the mid-call mobility case, re-INVITE is issued to the correspondent node informing its new IP address. This approach requires the SIP infrastructure to track the terminal mobility. Also, in case the application does not use SIP as signaling, this method cannot be used.

2.7. Middleware

On the Android platform, the ConnectivityManager class can be used to monitor network connections, send broadcast intents when network connectivity changes, attempt to fail over to another network when connectivity to a network is lost, and provide an API that allows applications to query the state of the available networks [1]. Although such service may alleviate the need to evolve the network for vertical handover, this approach prohibits the reuse of existing applications as such on the mobile platforms. The application programming becomes essentially platform dependent. A more fundamental limitation is that it is not a communication but local event notification facility, so lacks the capability for sensing the state of, or sending notification to, the remote end. Consequently, any follow-up action to the vertical handover is confined to the local device. A direct consequence is that the old flow is rendered useless in the ignorance of the sender. Also, as the receiver side (the application or the user) should re-initiate the flow, it may take additional time and the Quality of Experience (QoE) of the user may degrade visibly.

Table 1 is the summary of the existing mechanisms and their shortcomings when they deal with the UDP receiver vertical handover. In the next section, we design the UDP extension that can handle the vertical handover more elegantly.

3. Backward-compatible UDP extension

In this section, we discuss the design of the UDP that meets the principles stated in Section 1. Since its original specification [13], UDP (i.e., IP protocol number 17) has
not changed at all. It is a purposely simple protocol, and it is nearly impossible to introduce additional features to UDP. Its header is only eight octets long, with neither reserved bits nor incrementally definable option fields. UDP contrasts with its sibling protocol TCP, which has been richly evolved since the original specification [14]. The option field in TCP has been effectively harnessed for incremental evolution for unforeseen uses and newfound inadequacies. Although UDP is still a popular transport protocol second only to TCP, its limitations frequently require the introduction of extrinsic components to supplement it. The most representative example is the Real-Time Protocol (RTP) [15] designed to overcome the inadequacy in the UDP for applications exchanging real-time data.

Were it not for the backward compatibility issues, modifying UDP or replacing it with some advanced protocol would be a non-problem. But in practice, any such endeavor would be of little practical value. First, existing installed base of UDP-based applications will become non-interoperable. Second, persuading application programmers to move to a new API begs strong justifications and will take time if it ever succeeds. So the core of the UDP evolution problem is how to introduce the change in a user-transparent manner, and without the backward compatibility problems with legacy peers. The lesson from the TCP evolution is that we may achieve it through the use of options. Namely, check the UDP peer for the option processing capability, and if not, simply revert to the old way. But if both ends support the modified UDP, enable the use of options. And a specific option is turned on only when both ends agree on it. Below, we discuss how we can introduce options and achieve this desired mode of operation in UDP.

### 3.1. Signaling the compatibility

Above all, the question that we must ask about the UDP evolution is how we signal the peer UDP that we are the options-enhanced UDP in the first place. For the easiest answer, one could think of designating some port numbers for the signaling purpose, as there are more than a few reserved ones for UDP such as 0, 1023, 1024, 1109, etc. [16]. When receiving a datagram destined to this "signaling port(s)," UDP itself could decode it instead of forwarding it to any application. However, the standardization of such port numbers and registration with IANA itself would require considerable standardization efforts, and yet it would not completely prevent confusion between UDP peers due to ignorant applications and legacy UDPs inadvertently using the special ports. Next, one could consider changing the checksum to flag the enhanced capability. Unfortunately, there is no unused number in the UDP checksum space. So, altering the checksum in any way would cause the receiver to drop the UDP datagram, even when the checksum is "not used" (0x00 00). Moreover, a middlebox changing any value in the IP or UDP headers (e.g., Network Address Translation (NAT)) would cause the recalculation of the checksum, interfering with the intended signaling. Therefore, using the checksum field is out of the question.

The last candidate is the length field. It indicates the size of the 8-byte UDP header plus the payload length. So the minimum value is 8 (no payload), and any datagram with length smaller than 8 is silently discarded. First, we could think of using these unused lengths (0–7) for signaling purpose as it would not be confused with normal data length. However, it would cause a deployability problem for the modified UDP. The routers in the network generally do not look into the transport header, so the packet is unlikely to be dropped inside the network. But Linux (hence Android), Windows, and iOS all silently discard UDP datagrams with length below 8. Maybe we could ask all the OS's treat these strange length datagrams specially, not drop them but interpret the length value as some signaling message. However, even that would not help, since middleboxes such as firewalls and packet filters such as netfilter [17] can discard them before the datagrams even reach the UDP.

### 3.2. UDP length 8 and the eXtendable UDP

Use of UDP lengths either over or below 8 are problematic for our purpose, but 8 itself is special. It is neither a Martian length that can provoke middleboxes or filters before the datagram reaches the UDP code, nor a length used by any meaningful application that are supposed to exchange data. Using UDP length 8 with no data payload does not interfere with normal applications in any significant way either, since the worst case behavior would be that zero byte is read by the application. Indeed, it is considered normal that UDP receives a valid UDP datagram with a zero-length payload (Best Current Practice 145 [18], Section 3.6). In practice, the legacy UDPs on Windows and iOS do not even let the application know that a datagram of zero-length payload arrived. Those on Linux and Android do let the application know, but no data is sent to it. Either way, since no data is passed to the application, there is no danger of inserting unwanted bytes into the data stream for the receiver. So in this paper we propose to use UDP length 8 to signal to the peer UDP that it is the enhanced version, and can use options. Specifically, when and only when the UDP that first sends the empty datagram receives another empty datagram from the peer UDP that it can start using options (see Fig. 4). Namely, the UDP peer that needs to use the options will initiate the compatibility check. In case both UDP ends transmits the empty datagram to check the peer's capability almost concurrently, the datagrams can cross each other, and the bi-

### Table 1

Summary of vertical handover schemes.

<table>
<thead>
<tr>
<th>Layer</th>
<th>Scheme</th>
<th>Shortcomings</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Bonding interface</td>
<td>Static configuration</td>
</tr>
<tr>
<td>3</td>
<td>Logical interface</td>
<td>Administratively bounded</td>
</tr>
<tr>
<td></td>
<td>3GPP EPC</td>
<td>Administrative technology only</td>
</tr>
<tr>
<td>2.5</td>
<td>IEEE 802.21 (MIH)</td>
<td>Assist only</td>
</tr>
<tr>
<td>3</td>
<td>3GPP ANDSF</td>
<td>Administratively bounded</td>
</tr>
<tr>
<td>4</td>
<td>PMIP extension</td>
<td>Infrastructure required</td>
</tr>
<tr>
<td>5</td>
<td>SCTP DAR</td>
<td>Different service model</td>
</tr>
<tr>
<td></td>
<td>SIP DAR</td>
<td>Application should adopt SIP</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Infrastructure required</td>
</tr>
<tr>
<td></td>
<td>Middleware</td>
<td>ConnectivityManager</td>
</tr>
<tr>
<td></td>
<td>ConnectivityManager</td>
<td>Continuity loss</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Local repair only</td>
</tr>
</tbody>
</table>
directional compatibility check can be complete with only the two empty datagrams. In case the compatibility-checking empty datagram is not acknowledged, it will be retransmitted to a preconfigured number of times.

### 3.3. Option-carrying UDP format

Once the extended capability is confirmed by the exchange of length 8 packets, the next question to ask is how we exchange UDP options now. To tell the conclusion first, the UDP length 8 datagram itself cannot be used for such exchange. If it could, the two cases depicted in Fig. 2 would be the candidate formats. The first case (above) is where the IP total length is set to include the UDP options. The problem with this configuration is that the UDP length and the IP total length information do not agree with each other, and this datagram can be filtered in the middle of the network. In particular, the Robust Header Compression (ROHC) is frequently employed in 3G/4G cellular networks, but the ROHC standard explicitly states that the Length field of the UDP header must match the length field(s) of the preceding subheaders, e.g., the IP header [19]. The second configuration (below) is also problematic, not only because the ROHC stipulates that there must not be any padding after the UDP payload that is covered by the IP Length [19], but middleboxes such as NATs usually truncate the part that is not covered by the IP Length. In that case, the attached UDP options will not be conveyed to the receiving end.

The UDP options, therefore, should be carried on separate UDP datagrams. Fig. 3 shows the format of the option-carrying UDP datagram. In these datagrams, we include a 4-byte magic cookie that starts the UDP payload. The 4 bytes are 0x58 55 44 50 (“XUDP” in ASCII). In order to prevent a normal UDP datagram that happens to carry data that starts with 0x58 55 44 50, we use the stuffing that repeats the four bytes. For example, “XUDP” is replaced by “XUDPXUDP.” For those 4 bytes patterns that appear not in the starting position of the UDP payload, we do not perform stuffing. In the worst case, a normal UDP datagram can inflate by 4 bytes, with a very low probability that the first four bytes are exactly “XUDP”. When a UDP datagram arrives, the receiving UDP checks if the first four bytes is the magic cookie. If not, this is an ordinary data packet, so it is forwarded to the application. But if the datagram starts with the magic cookie, it checks the next 4 bytes to see if “XUDP” has been stuffed. If so, it removes the stuffing and forwards the payload to the application.

![Fig. 3. UDP option carrying packet.](image)

But if not, it parses the option, without forwarding any byte to the application.

To match the request and response in option exchanges, a 2-byte transaction ID (Xid) field is inserted in each option-carrying packet. This value can be any number, except that its first octet must not be 0x58, to guarantee that the aforementioned stuffing is not interfered. The UDP option takes the type-length-value (TLV) format, as in many other Internet protocols. We could expand the list of options as needs arise. We believe that 2 bytes are sufficient for the Type and Len fields except Val, which is of a variable length. Some options may not have Len and Val fields, as in TCP options. The Len field of an option gives the number of octets after the length field. Note that this length field is different from the one in the original UDP header. The original UDP header has length 8 plus the length of the options field including the magic cookie and the Xid. Currently, we assume that only a single option is carried by a datagram. It is for the ease of the explanation, but the format is already designed to be able to carry multiple options in the same datagram. If needed, multiple options can be carried in future.

As the readers may have noticed above, we will call this UDP that performs the compatibility check using the length 8 datagram followed by option transmission by the name of XUDP, since it is a UDP made eXtendable. Note that these components of XUDP do not break the header semantics of the original UDP protocol, making XUDP interoperable with legacy UDP. So we keep the IP protocol number 17 for XUDP. We remark that XUDP should be distinguished from other UDP “extensions” that are essentially a disparate protocol. For example, UDP-Lite [20] redefines the semantics of the length field to mean checksum coverage instead, so its protocol number is 136 instead of 17. One could also look at the adoption of options as the infusion of in-band signaling\(^1\) capability in the UDP, which its sibling TCP has long had.

\(\text{Fig. 2. Non-working cases.}\)

\(\text{Fig. 3. UDP option carrying packet.}\)

---

\(^{1}\) In TCP/IP literatures and in the socket interface, this type of signaling is called out-of-band. But as Stevens points out [21], in that the receiver must examine every byte looking for “XUDP” in the payload starting position, in-band signaling is a more appropriate term. But as far as the compatibility check before the options exchange is considered, it can be called out-of-band. In this paper, we choose the term in-band signaling to collectively describe these XUDP operations.
3.4. XUDP option signaling

Fig. 4 shows an example of the setup of the enhanced UDP session. The figure depicts the case where the compatibility check and the option processing are performed at the beginning, but they can be attempted any time during the lifetime of the given UDP flow. Moreover, they can be initiated by either party that wants to use options. The only restriction here is that the XUDP option processing can happen only after the compatibility check is complete. The options can be negotiated multiple times over the UDP flow lifetime, as needs arise. If there is an option that the receiving XUDP does not understand, it ignores the option so that the option is not used in the consequent conversation. Note that during this option signaling process, the normal UDP data stream is not suspended or interfered in any significant way. They can simply continue as in legacy UDP.

Finally, an aspect of XUDP signaling is that to address the case that the XUDP datagram (including the length-8 datagram used for compatibility check) is lost, the signaling messages must be acknowledged from the receiving peer. Otherwise, the XUDP should retransmit it up to a certain number of times, e.g. 3, before giving up. As in TCP, we let the maximum number of retries for the XUDP signaling messages be a system configuration parameter.

4. XUDP for vertical handover

The addition of option-carrying or in-band signaling capability opens a whole new horizon for the usability of UDP. In this section, we apply XUDP to the problem that we started with, i.e., maintaining the UDP service continuity across vertical handover, possibly in the absence of any mobility management infrastructure. We first discuss how XUDP can be harnessed to provide a solution to the problem. We also discuss briefly how the proposed scheme may work in the presence of existing infrastructure. Finally, we explain how the idea is easily extended to move UDP flows also between different mobile devices, and to let UDP use multiple path simultaneously.

Unlike TCP, UDP transport can be either unicast or multicast. Although XUDP can easily support multicast as well, we will limit our discussion to the unicast UDP flows below. This is because the vertical handovers that we experience today usually involve cellular networks, but generally the cellular networks do not support multicast.

4.1. XUDP options for vertical handover

For UDP service continuity over vertical handovers, we define a small set of XUDP options as shown in Table 2. For other XUDP use cases, more could be added in future. The options for the vertical handover serve as request or response. ACK and NACK are used to assist signaling interactions between XUDP peers (see Fig. 4). Their roles are similar to those in other protocols such as DHCP. We discuss the other options below in the description of the vertical handover preparation and execution steps.

4.2. Endpoint registration

When a UDP flow is established, the receiving endpoint performs the compatibility check first. If the sending end is XUDP-capable, the receiving end registers its secondary interfaces\(^2\) for possible vertical handovers. For this, the AD-DEND option is used. Contained in the Val field in the option are the interface address being registered and the primary interface information. If there are more than one backup interface, separate registration is done for each interface.

\(^2\) One of the issues in vertical handover is whether the mobile can have two or more interfaces active at the same time. For instance, 3GPP allows so called the dual-radio handover whereas the IEEE 802.21 only allows the single-radio handover. In the current design, we take the dual-radio approach since the signaling message transmission on the backup interface is minimal and so is the battery consumption by using mechanisms such as power-save modes if necessary. It is our future work to design the XUDP-based vertical handover under single-radio.
Table 2
XUDP options used to support vertical handover and their carrying values.

<table>
<thead>
<tr>
<th>Use</th>
<th>XUDP option</th>
<th>Val</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acknowledgement</td>
<td>ACK</td>
<td>–</td>
</tr>
<tr>
<td>Disapproval</td>
<td>NACK</td>
<td>–</td>
</tr>
<tr>
<td>Interface de/registration</td>
<td>ADDEND</td>
<td>IP, port (to be registered)</td>
</tr>
<tr>
<td></td>
<td>DELEND</td>
<td>IP, port (to be de-registered)</td>
</tr>
<tr>
<td>Ack. to ADDEND</td>
<td>ADDEND-ACK</td>
<td>IP, port (registered)</td>
</tr>
<tr>
<td>Ack. to DELEND</td>
<td>DELEND-ACK</td>
<td>IP, port (de-registered)</td>
</tr>
<tr>
<td>Flow switch</td>
<td>SWITCHTO</td>
<td>IP, port (target)</td>
</tr>
<tr>
<td>Primary change</td>
<td>SETPRIMAR Y</td>
<td>–</td>
</tr>
</tbody>
</table>

In this paper, we assume that a policy is given to the mobile device as to the number of interfaces to be registered and their priorities. For instance, a mobile can be configured to use multiple interfaces in the priority order of Wi-Fi → 4G → 3G on the availability basis. Namely, Wi-Fi is preferred to 3G and 4G, and 4G to 3G. For example, if Wi-Fi comes up while using other, we switch back to Wi-Fi. The policy can be much more complex, and specifying its possible forms is beyond the scope of this paper. Below, we will simply assume that there is such a priority over interfaces, given by a policy.

Using the primary and the secondary interface information provided in the ADDEND, the XUDP sender creates a logical coupling between the two interfaces by putting the secondary address in the same socket with the primary. Then the sender acknowledges ADDEND by sending ADDEND-ACK to the primary interface at the XUDP receiver. In the Val field, the registered address is confirmed. DELEND is to deregister a secondary interface, when it is no longer available or used at the XUDP receiver, or a new address is assigned to the interface. It removes the coupling between the secondary and the primary in the UDP sender socket. In case the primary interface is rendered unusable or even purged, SETPRIMARY can be used to change the primary interface designation as in SCTP. When it is issued from a previously secondary interface, the sending UDP ACKs any XUDP messages towards the new primary receiving interface instead. Since the primary interface is the target of the ACK from the UDP sender, the UDP receiver should always maintain a primary designated interface. Obviously, only the interfaces already registered as a secondary can be set primary.

The ADDEND-ACK should be ACKed itself, in order to prevent an illegitimate third party from registering its interface. This 3-way handshake is to cope with the possible loss of the ADDEND-ACK. If for any reason the ADDEND or the DELEND is not acceptable, it should be NACKed. For instance, a third party may attempt to ADDEND illegally its own interface for an existing UDP flow, for example to hijack it by issuing a SETPRIMARY from the interface later. As the XUDP sender will acknowledge this illegal ADDEND to the current primary, the legitimate user will see the unsolicited ADDEND-ACK, for which it will issue a NACK. Then the UDP sender should ignore the illegitimate ADDEND. When a NACK is sent in response to an ACK, the NACK itself should be ACKed. Otherwise, if the NACK is lost in the network, it would result in allowing the illegitimate hijacking without the ACK. Fig. 5 shows the call flows for successful and failed ADDEND attempts. XUDP will surely raise many security issues as well as provide new potentials. But we believe that exploring the potentials is first, on which we will focus below.

On mobile devices, the IP addresses for the interfaces may frequently be provisioned by the ISPs/carriers from a private IP address space. XUDP needs to cope with the situation because ADDEND and DELEND options carry IP addresses in the payload. To prevent private addresses from being registered at the XUDP sender after the ADDEND crosses a NAT box, the XUDP sender must take the translated IP address of the registering interface. So the XUDP sender registers the address appearing in the IP header instead. Note that the layer crossing between L3 and L4 as above is frequently done in the TCP/IP processing as can be seen in the use of IP addresses in the transport checksum computations ("pseudo header" [13]). The port number is processed likewise. The ADDEND-ACK confirms the registered endpoint. In case NAT is in use, this registered end can be different from the submitted endpoint in the ADDEND. The UDP receiving end must take note of the confirmed endpoint information in the ADDEND-ACK, since it may be used in DELEND later. Unlike ADDEND, the DELEND can be issued from other interfaces than is being deleted. So on the UDP receiver side, the socket has to record both the original and the translated endpoint information (see Section 4.4 for UDP socket augmentation for XUDP).

4.3. Vertical handover execution

A SWITCHTO option is central to the vertical handover support. It triggers the UDP flow to move to the designated interface in the Val field. For instance, it can happen when the primary interface has become unavailable (e.g. due to switch-off) or unusable (e.g. due to severe QoS degradation). When the primary interface comes back on, the movement from the secondary to the primary is also executed by a SWITCHTO. Fig. 6 shows the vertical handover executions triggered by the loss and the subsequent recovery of the primary interface. For simplicity of explanation, we will assume that there are only two interfaces, primary and a backup, in this paper. The extension to more than 2 interfaces is straightforward. The SWITCHTO can be issued on the primary interface as well as on a secondary interface.

4.4. UDP socket augmentation

As shown in Fig. 6, at both ends of the UDP data stream, the XUDP socket should be able to accommodate multiple endpoints. In Linux for example, we need to allocate one net_device structure for each interface and add its pointer in the existing UDP socket udpsock. The net_device structure is also modified for the XUDP receiving side to store the original (e.g. private) address before translation and the translated address confirmed by the XUDP sender.

Because a XUDP socket has multiple addresses, when a datagram arrives, the destination address is checked against the primary and the secondary addresses to find
the matching socket. On the sending UDP side, the signaling packet sent by the receiving UDP either from the primary or from the secondary interface is accepted. Other than that, the normal UDP receiver side check is performed to find the matching socket. Also for “connected” UDP sockets that check both the sender and the receiver side to match the socket, the receiver side check is similarly modified in XUDP. Usually it is the UDP receiver that connects, to filter out all but one sender. But in case the UDP sender connects, it sends to only one receiver. In case SWITCHTO is received, the XUDP sender executes connect again to change the mapping [22]. The connect can also be used to simplify the routing by choosing the local IP address to use for the given destination. XUDP also deals with this optimization that it may override it to use the interface of its own choice (e.g. a secondary interface).

4.5. Interface status notification

In order to execute the vertical handover, the receiving end XUDP needs to get alerted when the status of an interface changes (In the outgoing direction, however, the routing table is appropriately updated, so XUDP does not need to get involved.) To this end, we use standard facilities that are already in the kernel. In Linux for instance, we perform register_netdevice_notifier in the udp_init function, which triggers the notifier handler to inform UDP (and any interested parties in the kernel for that matter) whenever there is change in the interface status. For vertical handover execution, we are interested in two types of changes in the receiving interface: interface down and link loss detection. The former case for example happens when we turn off the Wi-Fi interface. An example of the latter on the other hand can be moving out of the access point (AP) coverage or AP power-off.

Table 3 summarizes what we need to examine upon the interface status change notification. When an interface reports NETDEV_CHANGE, it could have multiple causes. In this case, we need to look into the device structure to identify the exact source. If the flag has the IFF_RUNNING bit cleared, we determine that the interface went down. But if the signal was NETDEV_DOWN, it clearly means the device is no longer usable. An interface coming back up is checked similarly.

Upon these status changes in an interface, the notifier handler udp_netdev_event function returns the pointer to the device. Then XUDP identifies the socket structure that points to the device, and takes the required action. For example, if the down interface is the primary, then it issues SWITCHTO from the secondary interface towards the XUDP sender.

4.6. Working with existing frameworks

Since there can be other mechanisms in place to support vertical handover, XUDP should be designed to work with them harmoniously.
choose to use UDP (hence XUDP) as an Internet transport must employ mechanisms to prevent congestion collapse and to establish some degree of fairness with concurrent traffic [23].

5. Proof-of-concept implementation and performance evaluation

In this section, we discuss how our proof-of-concept prototype of XUDP is implemented and how it performs on real-life commercial networks that comprises both wireline and different types of wireless networks. We implemented XUDP on Google Nexus One Android smart phones and Linux laptops, by adding the XUDP code to the Linux and the Android kernels. We ran the prototype implementation between mobile and stationary devices connected across commercial ISPs and 3G cellular networks that dynamically assign private IP addresses and perform NAT and ROHC. XUDP should overcome such technical difficulties to be used universally, as we discussed in the introduction. We tested XUDP for various scenarios such as vertical handovers on one side, vertical handovers on both UDP ends, and multi-path UDP. Also we tested a backward compatible but XUDP-aware socket API. We will demonstrate below that XUDP accomplishes all these intended operations in the given environments.

5.1. XUDP pseudo-code

In most cases, a vertical handover is triggered by the change of state on an interface. XUDP requires to know such change to switch the interface and let the ongoing UDP flow(s) survive the disruption. Algorithm 1 shows how the XUDP entity is notified of the interface state changes. When a device fails, the socket structure table \((sk\_table)\) is searched through for any socket that uses the device as its primary interface (line 4). For such sockets, the client state \((C\_state)\) associated with the socket is set to \(SND\_SWITCHTO\) and a XUDP message is sent to the server so that it can switch to a secondary interface that has been registered by the client. XUDP maintains per-socket states to synchronize the two ends of the UDP flow as to the interface(s) to use, which we describe later on.

Algorithm 1. Interface event notification

\[
\begin{align*}
1: & \text{function} \; Event\_handler \; (dev, \; event) \\
2: & \text{if} \; event \; == \; DEV\_FAILURE \; \text{then} \\
3: & \; \; \; \text{for all} \; sk \; \in \; sk\_table \; \text{do} \\
4: & \; \; \; \; \text{if} \; sk.py \; == \; dev \; \triangleright \; sk.py \; \text{is the socket's primary interface} \\
5: & \; \; \; \; \; C\_state \; \leftarrow \; SND\_SWITCHTO \\
6: & \; \; \; \; \; \text{Send}\_XUDP(sk.O\_SWITCHTO)
\end{align*}
\]

Such XUDP messages as sent in line 6 of the previous code should be handled by the XUDP entity receiving them. Note that the receiving end could be either the server or the client. Whether at a server or at a client, Algo-

<table>
<thead>
<tr>
<th>Event</th>
<th>Device flag</th>
<th>Decision</th>
</tr>
</thead>
<tbody>
<tr>
<td>NETDEV_DOWN</td>
<td>-</td>
<td>Interface down</td>
</tr>
<tr>
<td>NETDEV_chg</td>
<td>-IPF_RUNNING</td>
<td>Interface up</td>
</tr>
<tr>
<td>NETDEV_UP</td>
<td>-IPF_RUNNING</td>
<td>Interface up</td>
</tr>
</tbody>
</table>

4.6.1. MIH and ANDSF

When the vertical handover takes place within an administrative domain, MIH or ANDSF could be working closely with higher layer mobility management protocols such as MIP (PMIP) or SIP mobility. The selection function can determine the next interface to use upon a vertical handover, and this policy decision should be respected. Although these standards have considered working primarily with L3 or L5 mobility management standards, there is no reason why it cannot serve XUDP (L4) as well. For instance, the selection function can assist the policy decision as to the number of interfaces and their priorities can be given as an input to XUDP. When implemented, MIH/ANDSF SAPs are functions (+callbacks) in kernel. XUDP would invoke them as service. In the future, we will investigate if and how we can interface XUDP with MIH or ANDSF, and utilize the assistance provided by these frameworks in their presence.

4.6.2. SIP mobility

The application might be using SIP signaling protocol, which can cope with mid-call vertical mobility through re-INVITE from the new interface. Then a media path will be created to the new interface. What is desirable here is that XUDP does not interfere with the media plane control by SIP. We will discuss in Section 5.6 how the application can turn off XUDP when other possibly conflicting vertical mobility execution mechanism such as the SIP mobility is in charge. But even if such coordination is lacking, XUDP would not significantly stymie the SIP control. Suppose the primary interface loses link, and both the SIP mobility and XUDP take action on the data plane. If the SIP signaling is first executed, there is nothing that XUDP should do. XUDP will find that the socket entry for the flow has already gone. On the other hand, if the XUDP reacts to the change of the interface first, the flow switch will happen on the socket, but the SIP mobility will simply purge it and create a new one.

4.6.3. TCP-friendliness

One might think that in case XUDP becomes popular for its strengths, it could cause congestion control issues since XUDP is a “transparent” superset of UDP, which only assists with the vertical handover aspects. The concern is valid, but it is already addressed by TCP-friendly UDP usage guidelines put forth by the IETF in the Best Current Practice (BCP) 145 [23]. In essence, the traditional approach to the TCP-friendliness issue with UDP applies in exactly the same way here as well, in other words the problem should be addressed by application, not by UDP itself. As the BCP 145 puts it, the applications and upper-layer protocols that...
Algorithm 2. XUDP receiver processing

```
7: function udp_rcv(pkt) ▷ pkt: arriving UDP
datagram
8:  uh ← udp_hdr(pkt)
9:  ulen ← uh.len ▷ Extract UDP length
10: if ulen == 8 then ▷ XUDP compat. check
    11:    pktType ← COMPCHK
12:    else if ulen > 8 then ▷ XUDP header begins.
    13:       Xuh ← Xudp_hdr(pkt) ▷ Xuh: XUDP
    14:       if Xuh.magic_cookie == "XUDP" then
    15:          pktType ← XUDP
    16:       else
    17:          pktType ← UDP
    18: else ▷ Illegitimate length: abort processing
    19:    Free pkt; return 0
20:  sk ← udp_lookup_skb(pkt) ▷ Dispatch on
    21:  if sk == NULL then ▷ the packet type
    22:      Return an ICMP error to the sender and free
    23:      pkt; return 0
24:  if pktType == COMPCHK then ▷ Dispatch on
    25:      Proc_COMPCHK(sk.pkt)
26:  else if pktType == XUDP then ▷ the packet type
    27:      Proc_XUDP(sk.pkt)
28:  else if pktType == UDP then ▷
    29:      Proc_UDP(sk.pkt)
```

Algorithm 3. Socket matching

```
31: function udp_lookup_skb(pkt)
32:    for all sk ∈ sk_table do
33:       if sk does not bind dport then ▷ Packet’s
dst port should match
34:          Continue
35:       if sk.IF_tab != NULL ▷ Only XUDP client
          maintains IF_tab
36:          for all addr ∈ sk.IF_tab do
37:             if addr == addr then
38:                return sk ▷ Client processing
39:       if isConn(sk) then sk ▷ connected to a
client IP addr
40:          if Xuh.type == O_ADDEND&&&Xuh.val.IP2 == sk.py
31:             then
42:                return sk
43:          else ▷ sk only binds dport
44:             return sk
45:    return NULL
```

Algorithm 4 shows how a legacy UDP packet is processed by XUDP. Note that any XUDP operation can only begin after some legacy UDP datagrams arrive at the client. As discussed above, c_state is associated with a client socket, and it is initialized to NULL when the socket is created. So before the XUDP interface registrations, only the condi-

The udp_lookup_skb function that finds the matching socket structure on the second step of the XUDP processing is described in Algorithm 3. In the algorithm, the destination address (daddr) and port (dport) are extracted from the packet, which is not elaborated for brevity. From the socket structures table maintained by the kernel, XUDP should find a socket that matches the arriving packet. For a match, the socket should be first bound to the same port number that the packet is destined to (line 33). At the client side, the packet’s destination address should match one of the IP addresses in the socket’s interface table IF_tab, with which addresses are registered by XUDP. On the client side, the processing ends here. If a qualifying socket is found, it is returned. On the server side, the check in line 35 fails, and the processing continues to line 39. If the server has “connected” to a specific destination IP address in addition to the bound port, it should be to the primary IP address of the client. If the secondary address of the client transmitted a XUDP packet while the primary is still in the active state, then it must contain an ADDEND. In that case, the validation is completed by checking the primary IP address that the XUDP client attached (Xuh.val.IP2, line 40). This IP address is carried in the XUDP message sent by the secondary interface is the address of the primary address. As the server keeps the primary address in the socket, it can match Xuh.val.IP2 with it. If the two match, the socket is returned.
tion in line 67 matches when the legacy UDP datagrams arrive. Consequently, the client sends a compatibility check packet to the server after setting the C_state to SND_COMPCHECK. Once the client socket state is changed to SND_COMPCHECK, the attempts to check the XUDP compatibility of the server are made up to a certain threshold, which is 10 in the current prototype implementation (line 48). For each arriving legacy UDP datagram, a compatibility check packet is sent (line 49). In future, this logic will be replaced by an elapsed time check as the speed of the UDP flow will increase and the number of legacy UDP datagrams before a timely response by the server can exceed such a low threshold. But in the current implementation, the response, an XUDP message carrying O_COMPATCHK, arrives before the threshold is crossed. If the server does not respond to the check until the threshold is exceeded, however, the client declares that the server is not XUDP compatible (line 51). Only the legacy UDP can be used for the UDP flow then.

If legacy UDP datagrams still arrive when the client is in either SND_ADDEND, SND_SWITCHTO, or SND_NACK, it could be because the XUDP packet carrying the corresponding messages (ADDEND, SWITCHTO, NACK) was lost. So for these cases, Snd_XUDP is invoked for retransmission (lines 54, 57, 65). If a new interface that has not yet been registered becomes available when both peers are synchronized as to the other interfaces (SND_NEGOTIATED), it should be notified to the server. So the client sends O_ADDEND to the server after changing the state to SND_ADDEND (line 62).

Algorithm 4. Legacy UDP datagram processing

```
45: function Proc_UDP(sk,pkt)
46:   if C_state == SND_COMPCHECK then
47:     cnt ← cnt + 1
48:     if cnt ⪯ COMP_THRESH then  \textbullet Default is 10
49:       Snd_XUDP(sk,pkt,O_COMPATCHK)
50:     else
51:       C_state ← SND_NOCOMPATIBLE
52:   else if C_state == SND_ADDEND then
53:       Snd_XUDP(sk,pkt,O_ADDEND)
54:   else if C_state == SND_SWITCHTO then
55:       Snd_XUDP(sk,pkt,O_SWITCHTO)
56:   else if C_state == SND_NEGOCIATED then
57:     if a new interface not registered is turned on then
58:       C_state ← SND_ADDEND
59:       Snd_XUDP(sk,pkt,O_ADDEND)
60:   else if C_state == SND_NACK then
61:     Send_XUDP(sk,pkt,O_NACK)
62:   else if C_state ≠ NOCOMPATIBLE then
63:     C_state ← SND_COMPATCHK
64:     Snd_XUDP(sk,pkt,O_COMPATCHK)
```

Algorithm 5 shows the compatibility check logic at a XUDP capable receiver. The client that initiated the check should be in SND_COMPCHK after sending the compatibility check packet. The arrival of a compatibility check packet to such client means that the check is complete, and XUDP can be used between the peers. So the client proceeds to add secondary interfaces through O_ADDEND (line 73) after changing its state to SND_ADDEND, marking it transmitted an ADDEND request on this socket. When the receiver is not in SND_COMPATCHK state yet, however, the arrival of the check packet implies that the peer initiated it, so the receiver simply responds with a O_COMPATCHK, acknowledging its XUDP compatibility.

Algorithm 5. Compatibility check

```
70: function Process_COMPATCHK(sk,pkt)
71:   if C_state == SND_COMPATCHK then
72:     C_state ← SND_ADDEND
73:     Send_XUDP(sk,pkt,O_ADDEND)
74:   else
75:     C_state ← SND_COMPATCHK
76:     Send_XUDP(sk,pkt,O_COMPATCHK)
```

Algorithm 6 shows how a XUDP compatible receiver processes an incoming XUDP datagram. The first case is when the XUDP message type is ADDEND_ACK, which is sent by the server. Note that as the ADDEND request is initiated by the client, the client state should be SND_ADDEND upon receiving this message. Moreover, the IP address carried in the ADDEND_ACK should be the address specified in the preceding ADDEND request or the NAT-translated global address for the originally requested address. The address is registered in the aforementioned sk.if_tab structure, and O_ACK is sent in response (see Fig. 5, left). If all interfaces have been registered, then the client socket state is marked NEGOTIATED. Otherwise, the next interface to be registered is ADDEND’ed (line 86). If the ADDEND_ACK arrived when there is no outstanding ADDEND, it is NACK’ed (line 89).

When an ADDEND message arrives at the server (line 91), it validates that the request is from a secondary interface for which the primary address (Xuh.val.IP2) has been registered (line 92). If so, the addresses are temporarily registered, and the ADDEND_ACK is sent in reply. The registration is temporary because it will be complete only when the final ACK arrives from the client. When a SWITCHTO is issued by the client, the server needs to change the destination to a secondary IP address (line 98). When an ACK arrives when the server has been in SND_ADDEND_ACK, the temporarily registered addresses in line 94 are validated. But if the ACK is for a previous NACK, it means that we are on the client side. In this case, the server acknowledges the NACK, and the client state returns to normal, i.e., NEGOTIATED. Finally, when a NACK arrives from the client, the server checks if it has been in SND_ADDEND_ACK. This can only happen when the ADDEND_ACK was sent to a wrong client (Fig. 5, right). In this case, the server state for the client is restored to NEGOTIATED.
Algorithm 6. XUDP packet reception

```java
77: function Proc_XUDP(skpkt)
78:
79: if Xuh.type == ADDSEND_ACK then
80: if C_flag == SND_ADDSEND & Xuh.val.IP1 indicates requested IP addresses bf then
81: Register Xuh.val.IP1 to sk.IF_tab
82: Send_XUDP(sk,O_ACK)
83: if all interfaces registered then
84: C_flag ← NEGOCIATED
85: else
86: Send_XUDP(sk,O_ADDSEND) ▷ Send another
87: else
88: C_flag ← SND_NACK
89: Send_XUDP(sk,O_NACK)
90:
91: else if Xuh.type == ADDSEND then
92: if Xuh.val.IP2 indicates previously registered primary IP addr then
93: S_flag ← SND_ADDSEND_ACK
94: Temporarily add Xuh.val.IP1 and Xuh.val.IP2 to sk
95: Send_XUDP(sk,O_ADDSEND_ACK)
96:
97: else if Xuh.type == SWITCHTO then
98: Set a registered secondary IP address as the new primary
99:
100: else if Xuh.type == ACK then
101: if S_flag == SND_ADDSEND_ACK then
102: S_flag ← NEGOCIATED ▷ Validated
103: else if C_flag == SND_NACK then
104: C_flag ← NEGOCIATED
105:
106: else if Xuh.type == NACK then
107: if S_flag == SND_ADDSEND_ACK then
108: Deny the IP address which client requested.
109: S_flag ← NEGOCIATED
```

Below, we show the dynamics and performance of the XUDP when the vertical handover takes place over commercial and enterprise networks including 3G cellular, Wi-Fi, and wireline Internet. We look at the following cases: vertical handover on one side, vertical handover on both sides, and UDP flow delivery over multiple paths. These are all handled elegantly by XUDP code that we have described above.

5.2. Vertical handover on one side only

The most frequent vertical handover scenario is where the receiving end of the UDP flow changes interface, as a mobile device changes the mode of wireless access. To test XUDP for this case, we used a Google Nexus One smartphone where we enhanced its Android OS with XUDP. We set its Wi-Fi interface as the primary interface, and 3G as the secondary. For this experiment, we start a UDP flow from a server connected to a commercial wireline ISP network, towards the Nexus One smart phone connected by the Wi-Fi and/or the 3G cellular technology. The UDP flow carries a constant bit rate (CBR) stream transmitted at the rate of 100 packets/s, where the packet size is constant 1 KB. To reach the primary interface at the receiver, the UDP stream travels the wireline network of the ISP, except for the last hop, which is Wi-Fi. For the secondary, the stream exits the wireline network and enters the 3G network of the ISP. Then it is delivered over the 3G connection at the last hop. Fig. 7 depicts the experiment configuration. Both the 3G and the Wi-Fi interfaces obtain a class-A private IP address from the ISP through DHCP, such as 10.97.44.250.

There are two subcases where the vertical handover takes place on the receiver side. First, the mobile itself can turn off the primary interface. For example, the mobile user can disconnect the link to an open Wi-Fi access service for fear of security violation. In this case, the vertical handover is internally initiated by the mobile. Second, the wireless link can cease to serve the mobile. For example, it can happen when the user walks out of her office where the enterprise Wi-Fi connection is provided, and exits the building and gets to have only the cellular connectivity. In this case, the vertical handover has an external trigger. Below, we discuss how XUDP copes with the two realistic scenarios.

5.2.1. Handover initiated by mobile device

With the configuration given in Fig. 7, here we investigate the dynamics of XUDP when the smart phone itself cuts the primary Wi-Fi interface by disassociating with the AP. In the given configuration, the UDP flow from the server on the ISP network is initially delivered to the Nexus One smartphone through its Wi-Fi interface. Then the interface is turned off using the smart phone configuration facility. As we described in Algorithm 4, the arrival of the first few legacy UDP datagrams from the server triggers the registration of the secondary (3G) interface through XUDP, which remains dormant until the primary (Wi-Fi) interface is turned off. Fig. 8 visualizes the packet trace observed at the smart phone for the described scenario. In the
5.2.2. Handover triggered by wireless link loss

If the loss of Wi-Fi access is due to an extrinsic event, such as moving out of AP coverage or AP powering off, its detection can take longer. Few users would want to give up a Wi-Fi access just because of short-term link quality degradation. So the interface has to monitor the quality of the link up to a certain threshold to decide that the link is broken, not temporarily unavailable due to bad channel. In general, this is a policy or user preference issue, because for a vertical handover to be made for QoS reason, it should be the policy that sets the threshold for packet losses. In Fig. 9, we show the case where we leave the decision to the default configuration of the Nexus One phone. Just before \( t_1 = 16 \), we turn off the AP. The smart phone decides that the link to the AP is broken at around \( t_2 = 26.138 \). This \texttt{SWITCHTO} delivered to the server triggers it to transmit packets towards the receiver’s 3G interface instead, which begin to arrive at the Nexus One phone at \( t_3 = 27.946 \). Meanwhile, we power on the AP at around \( t_4 = 28 \), and it leads to the transmission of another \texttt{SWITCHTO} that causes the UDP flow to shift back to the Wi-Fi interface again. Immediately after the second \texttt{SWITCHTO}, outstanding UDP datagrams at the time of the switch still arrive through the 3G path, but later packets begin to arrive at the Wi-Fi interface at \( t_5 = 29.701 \). In the first switch from Wi-Fi to 3G, 1,210 packets were lost, but in the switch from 3G to Wi-Fi, only 3 were lost. Note that the asymmetry arises from the conservativeness in the decision on the loss of the wireless link (Wi-Fi here) to avoid spurious vertical handovers. Upon the new availability of the Wi-Fi link that used to be off, however, the switch back is almost immediate. In Section 5.6, we discuss how the application can control the switch-over threshold through a transparent but enhanced API so that the wireless link unavailability is more quickly determined and more critical applications can make the vertical handover earlier.

5.3. Handovers on both ends

In this experiment, we start by letting two Nexus Ones configured with Wi-Fi and 3G interfaces talk to each other over UDP. Namely, there exist two UDP flows in opposite directions. As before, these UDP flows trigger the registrations of the secondary interfaces as soon as they start. Namely, to receive the incoming flow upon possible vertical handovers, the phones \textsc{addEnd} their 3G interfaces to the server.

On each flow, we carry a CBR stream. Then we arbitrarily switch on and off the primary interface at either end. The switch is made through the configuration facility on the smart phone, so the detection of the lost wireless interface is immediate. One technical difficulty of this experiment is that it is essentially the peer-to-peer communication, but all the interfaces on these phones get private addresses from the service provider. As any other peer-to-peer application running across two NATs, therefore, we need for this experiment a Traversal Using Relays around NAT (TURN) [24] style server to relay traffic between these interfaces. So we put a relay server on the ISP, with a public IP address. For each UDP flow, it creates two XUDP sockets used respectively by the two UDP peers,
between which the UDP datagrams are relayed. In order to punch a hole to their respective NAT, the smart phones first transmit a separate UDP flow to the relay server. Using the already created sockets on the relay server, the flows are each relayed to the other smart phone. We remark that the XUDP functionality is not required on the relay server. Once the legacy UDP flow path is open between the two UDP peers via the help of the relay, XUDP works transparently between the peers. One of the strong points of XUDP is that it is an end-to-end solution, and it works over the middle boxes such as NATs, ROHC compressors, and relays.

In the experiment configuration as shown in Fig. 10, the two smart phones start the conversation using the Wi-Fi interfaces, but individually switch to the 3G and then revert to the Wi-Fi. In particular, the figure depicts the situation where the smart phone 1 (SP1) uses Wi-Fi and the smart phone 2 (SP2), 3G. Namely, it corresponds to the time duration between $t_1 = 9$ and $t_2 = 14$ in Fig. 11, where the packet traces received by the smart phones are shown. In the experiment, we start SP1 a few seconds earlier than SP2, so when SP2 starts the receiving application it begins to receive the packets with the sequence number greater than 500. So the time origin in the graph represents the instant that SP2 starts.

The most noticeable impact due to a vertical handover is shown upon the SP1’s switch to the 3G at $t_2 = 14$. Not only the reception on SP1 has a hiatus (lower trace), but also the reception on SP2 does. This is natural because both the transmission and the reception at SP1 should shift to the 3G interface. Other than this handover, the other three that are performed over the duration of the trace leads to relatively lighter impacts. The first handover by SP2 to 3G at around $t_1 = 9$ for instance takes only about 200 ms during which about 10 packets are lost. It is almost unnoticeable at the given scale. The other two switches are similar. Again, we also notice that the packet delivery performance on the Wi-Fi interface is better than on the 3G interface. But we can conclude that XUDP preserves the UDP flow continuity through the vertical handovers that occur on both ends of the UDP flows.

5.4. QoS-triggered vertical handover

So far, we considered the vertical handover that takes place as a result of losing an endpoint being used as the primary interface. In this section, we investigate the XUDP dynamics in a scenario where the handover is triggered by QoS monitoring. For example, a cheaper access mode such as a public Wi-Fi connection may not meet the requirement of a user application. The user may want to continue the application although it may be costlier. For such scenario, we can define a XUDP option that lets the XUDP client to execute a vertical handover if a user-specified QoS requirement is violated (How the user-requested QoS parameters can be conveyed to the XUDP is described in Section 5.6).

Fig. 12 shows a typical call flow in the QoS-triggered vertical handover implemented by XUDP. The QoS of the data reception on the current primary interface $rcv_{if1}$ is monitored by the XUDP receiver using the arriving packets if the user requested the QoS-triggered vertical handover. Meanwhile, the secondary interface $rcv_{if2}$ has to monitor the quality of the path between the sender and itself. Since no data packet is flowing on the path, it has to rely on some other technique. To that end, it periodically sends a pair of XUDP messages with an option called QoS_PROBE. The option carries two parameters: the size $n$ and the number $k$ of response(s). The XUDP sender responds to the XUDP message by sending the requested number of messages of the specified size back-to-back. The response to the QoS_PROBE is called the QoS_PROBE_RESP. By using $n \geq 2$, the probing interface can get a number of inter-packet gaps. Using the gaps, the secondary interface can keep track of QoS metrics of the path between the sender interface and itself such as throughput and delay. In particular, the technique implemented in the current XUDP is the Packet Pair/Train Dispersion (PPTD) technique [25], but other QoS measurement techniques could be employed in future.

When the QoS requirement is violated on the primary path, the XUDP receiver decides whether to move to a secondary interface based on the periodic measurement data on the secondary interfaces. If there is a better secondary interface and if the policy allows, it issues the SWITCHTO, and the XUDP sender changes the receiver endpoints. When the primary interface is changed, the secondaries are monitored as before. In particular, if the previous primary interface is monitored, and when its QoS metric

![Fig. 10. Two smart phones (SP1 and SP2) dynamically switching networks during two-way conversation.](image)

![Fig. 11. Received packet traces at the smart phones.](image)
meets the requirement, it is restored as the primary through another SWITCHTO.

We implemented the mechanisms discussed above, in particular the QoS_PROBE and QoS_PROBE_RESP options in the Android implementation of XUDP. In a real-life experiment using the prototype, we observe the dynamics of the QoS-triggered vertical handover. Again, the sender transmits 100 packets of size 1 KB every second, and in this experiment we assume that the receiver sets the minimum acceptable throughput at the primary interface at 80% of the transmitted rate, or 640 Kbps. Fig. 13 shows the packet trace captured at the XUDP receiver in this experiment. As before, we start with two interfaces. Here, we set the 3G as the primary interface, and Wi-Fi as the secondary.

At approximately $t_1 = 153.30$, the 3G bandwidth is measured 512 Kbps, which is below the minimum 640 Kbps. At this time, the secondary Wi-Fi interface is measured to have 5.347 Mbps according to the probing. Thus the XUDP receiver sends a SWITCHTO to the XUDP sender, requesting the switch of the receiving interface. When the XUDP sender switches, the 3G interface status changes to secondary, and the monitoring begins on it using the PPTD technique. At $t_2 = 157.66$, the periodic measurement finally reports 769 Kbps, above the threshold. Then the XUDP receiver issues another SWITCHTO to make the 3G interface the primary again. Fig. 14 shows the QoS monitoring process. The "3G-primary" is the throughput measured of the 3G (primary) interface via the arriving packets. As soon as the Wi-Fi becomes the primary interface, the throughput measured on the XUDP receiver surges over 1 Mbps, but this is the sum of the throughput on the fast Wi-Fi path and that on the 3G path that momentarily delivered packets that are outstanding at the time of the switch. We marked the background throughput check by the 3G (secondary) interface while the Wi-Fi is the primary by "3G-secondary." It shows that 154 through 156 s the measured throughput is below the required minimum. But at 157 s, the measured throughput exceeds the required minimum, so the XUDP receiver issues the second SWITCHTO and the 3G interface becomes the primary again. Our implementation immediately changes the receiving interface upon a single violation of the QoS requirement, but in practice it could be configured by the policy provided by the user. Also, other QoS metrics than the throughput could be specified by the policy.

5.5. Multi-path UDP

While there has been active discussions on SCTP and TCP for multi-path environment [26,27], there is a dearth of work on multipath UDP. To the best of our knowledge, there is only one: Yabandeh et al. [28] attempts to improve UDP performance in the multi-path context. In this section, we show that XUDP easily lends itself to enabling the multi-path delivery in UDP. In order to direct the sender to use multiple paths simultaneously, we define a simple option. Specifically, an option called SPLITTO lets XUDP split the data flow into 2 or more paths leading to the same number of interfaces at the XUDP receiver. As a parameter, the XUDP message carries the additional interface at which the new path should be made to. As in the ADDEND case, the message also carries the identity of the primary interface for validation purpose as well. With the three-way handshake between the server and the client, additional paths can be set up on needs basis. The current implementation maps a path to a interface in one-to-one manner, but other splitting modes could be added in future by defining other options. Expected applications of multi-path UDP delivery include offloading and multi-path diversity for QoS improvement etc.

In the following experiment, XUDP creates two paths between the UDP sender on the ISP network and the smartphone using the SPLITTO option: one through the 3G network and the other through the Wi-Fi. The primary path is the 3G, on which we send the packet stream at 1 K packets/s, or 8 Mbps, which the 3G link cannot keep up with. So the UDP throughput is limited by the 3G link throughput. Then we use the secondary path in addition to the primary. For this experiment, we programmed the UDP sender to alternate packets between the two paths. Fig. 15 shows the throughput result that we obtain in the real-life setting as used in previous experiments.

![Fig. 12. A typical QoS-triggered vertical handover scenario.](image)
In the figure, we can see that the throughput via the 3G path is mostly less than 1 Mbps, which is one eighth of the desired throughput. But at around $t_1 = 22$, the Wi-Fi interface on the smart phone starts to be used since at $t_2 = 22.628$, the smart phone sends the SPLITTO to the sender, specifying the Wi-Fi interface. As the receiver begins to receive packets on the Wi-Fi interface in addition to the 3G, the throughput jumps to over 4 Mbps. Although still just over half the desired throughput, the UDP client can download four times as much data as before. So the addition of the Wi-Fi interface can help improve the quality of experience (QoE) of the user. If other interfaces are available on the client, more paths can be created iteratively. Also, using the multipath can be created for opportunistic offloading, where the additional paths are created for taking some load off the typically congested cellular path. Setting aside issues such as out-of-order delivery, we argue that XUDP can provide a good basis for opportunistic offloading, whose potential is rhetorically demonstrated by this experiment.

5.6. Backward compatible XUDP API

One of the design principles of XUDP is that it should maintain the user transparency. It means that the API for XUDP should not be different from that of the legacy UDP. For instance, the `send` socket interface.

```
send(int s, const void *msg, size_t len, int flags);
```

should be the same for both UDP and XUDP. But for those XUDP-aware applications, we can enhance the traditional API to let them tap into the extra functionalities provided by XUDP, while maintaining the syntactic uniformity. We note that the additional XUDP functionalities are essentially signaling capabilities: signaling between the two UDP instances, and signaling between the application and the UDP. So far, UDP has not had any signaling. Therefore, the `MSG_OOB` flag for the socket interface has been used for TCP only, as the urgent mode is useful for some applications. For XUDP, we can use the flag for the signaling between the application and the UDP. Note here that we only add a usage for the exiting `MSG_OOB` flag formerly unused by legacy UDP. The application program that wants to direct the XUDP specifies the flag, and gives the control data as a parameter. For example, the `send` call above with the `MSG_OOB` flag can contain the XUDP option data in the `msg`. This `msg` can be sent from the application to the local XUDP instance, or can be translated into a XUDP option and sent to the remote XUDP end. On Linux, the `MSG_OOB` flag can be used in the socket API as discussed above.
On Android, however, apps are programmed in Java, and we need an interface program between the Java app and the C program that uses the MSG_OOB flag in the socket calls. After the C program is made into a native library, the user app can call it through the interface program.

We take an example of the application communicating with the local XUDP through the MSG_OOB. Recollect that in Fig. 9, we wanted to control the UDP behavior so that the SWITCHTO can happen earlier. In this experiment, the application directs the XUDP to switch to the alternative interface if the primary interface is not delivering packets for more than T seconds. Fig. 16 shows the result for setting T = 1 through the XUDP API from the receiving app on Nexus One, switching much earlier than in Fig. 9.

A second example usage of the API could be disabling XUDP option transmission as we discussed in Section 4.6.2. An application that uses SIP signaling for mobility management can issue the control message to turn XUDP off using a socket call such as send on the XUDP socket, with the MSG_OOB flag set.

6. Conclusion

Evolving UDP has been considered nearly impossible due to its extreme simplicity. In this paper, we show that there is an exit from the situation, so that UDP behavior is rendered modifiable while the backward compatibility is maintained. This opens a whole new horizon for UDP usability. UDP can thus be a feature-rich protocol that can solve the given problem with minimal effort. Finally, we also show that the evolved UDP can be used either transparently or explicitly since the applications can use the same API.

References


Jiung Yu received the B.E. and M.E. degrees in 2008 and 2010, respectively, from Korea University, Seoul, Korea, where he is currently working toward the Ph.D. degree. His research interests include software defined radio and mobile computing.

Jiun Ha received the B.E. and M.E. degrees in 2008 and 2010, respectively, from Korea University, Seoul, Korea, where he is currently working toward the Ph.D. degree. His research interests include software defined radio and mobile computing.
Byungjo Kim received the B.E. degrees in 2011 from Hongik University, Seoul, Korea, where he is currently working toward the M.E. degree. His research interests include vehicular network, localization, IoT, and mobile computing.

Hyogon Kim received the B.E. and M.E. degrees from Seoul National University, Seoul, Korea, in 1987 and 1989, respectively, and the Ph.D. degree from the University of Pennsylvania, Philadelphia, in 1995. He is currently a Professor with Korea University, Seoul. Prior to joining Korea University, he was a Research Scientist with Bell Communications Laboratory (Bellcore). His research interests include Internet protocols and applications, vehicular networking, software radio, and network security.