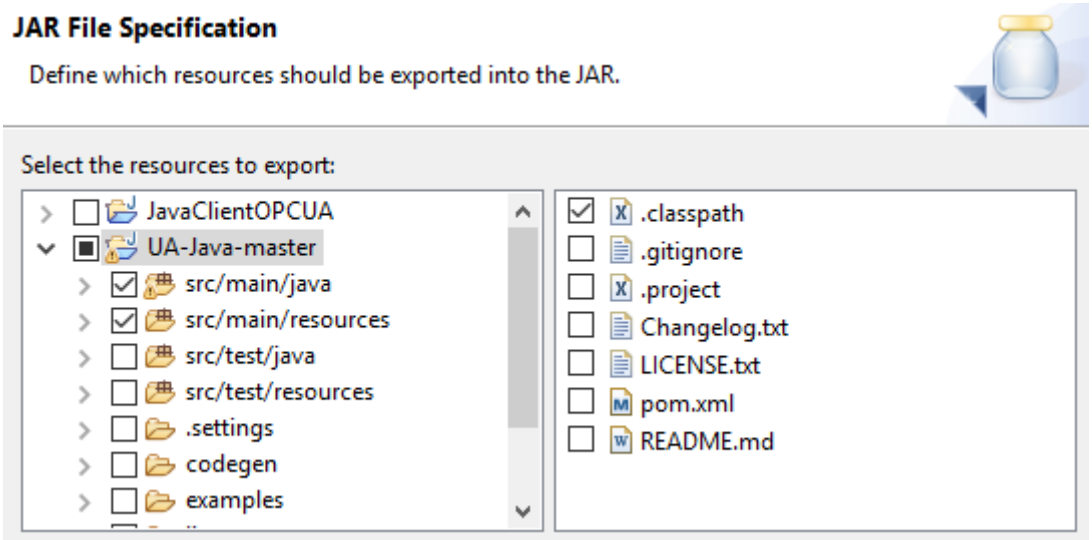
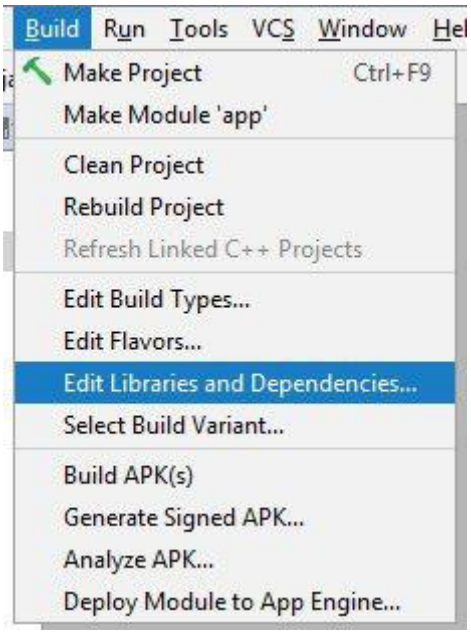
CONFIGURAZIONE AMBIENTE DI SVILUPPO

Per la realizzazione del progetto è stato utilizzato l'ambiente di sviluppo Android Studio 3.1.3, ambiente ormai ufficiale per lo sviluppo di applicativi nativi per Android sostituendo così Eclipse. Lo stack OPC UA fornito da OPC Foundation è disponibile per Eclipse sotto forma di progetto Maven (al seguente link: https://github.com/OPCFoundation/UA-Java). Questo ha causato inizialmente diverse problematiche in quanto non è stato possibile importare direttamente tale stack come libreria esterna da integrare al progetto. Si è resa quindi necessaria un'operazione preliminare per rendere compatibile questo stack con l'ambiente di sviluppo utilizzato. In particolare, è stato utilizzato l'ambiente di sviluppo Eclipse per la compilazione e l'esportazione dello stack in file .jar il quale è stato successivamente aggiunto come libreria esterna nel progetto su Android Studio. A seguire vi sono illustrati i passi seguiti per fare quanto detto sopra.

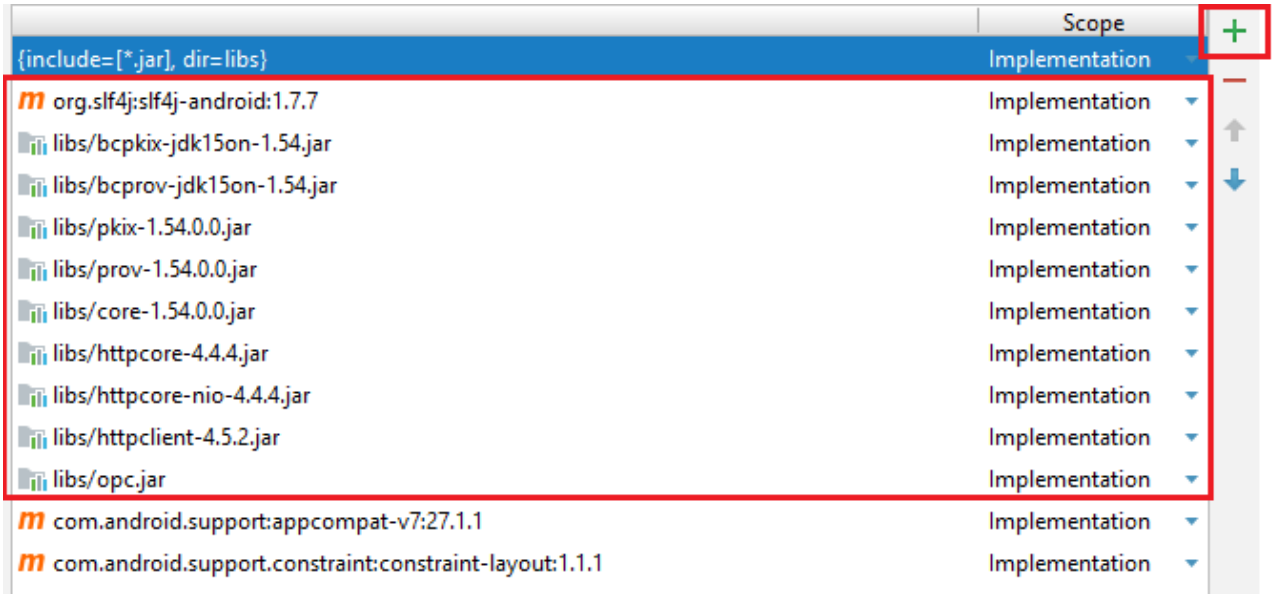
Una volta importato il progetto Maven fornito da OPC Foundation su Eclipse, dalla finestra JAR Export si è scelto di compilare ed esportare solo le cartelle che nella figura sottostante sono affiancate dalla casella spuntata.



Una volta reso disponibile il file .jar, si è passati ad Android Studio dove lo step successivo è stato aggiungere tale file come dipendenza esterna al progetto, mediante il menù a tendina mostrato in figura.



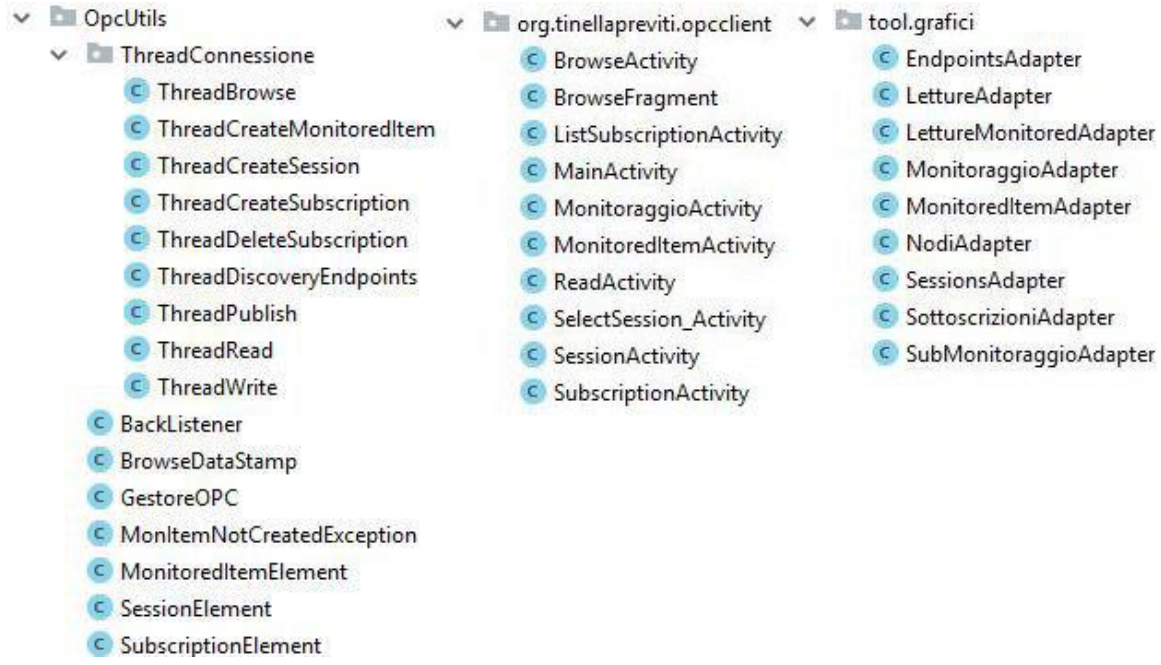
Insieme al file opc.jar e gli altri mostrati in figura, è stato necessario importare anche una dipendenza Maven evidenziata nello screen sottostante rispettando anche l'ordine di compilazione.



DESCRIZIONE DEL PROGETTO

Il progetto creato su Android Studio è compatibile con le versioni di Android che vanno dalla versione 4.0.3 (API 15) in poi, scelta adottata per avere un supporto quasi completo di tutti i dispositivi presenti sul mercato senza precludersi la disponibilità delle funzionalità più recenti di Android.

Le classi create nell'applicativo sono state strutturate come illustrato nella figura sottostante.

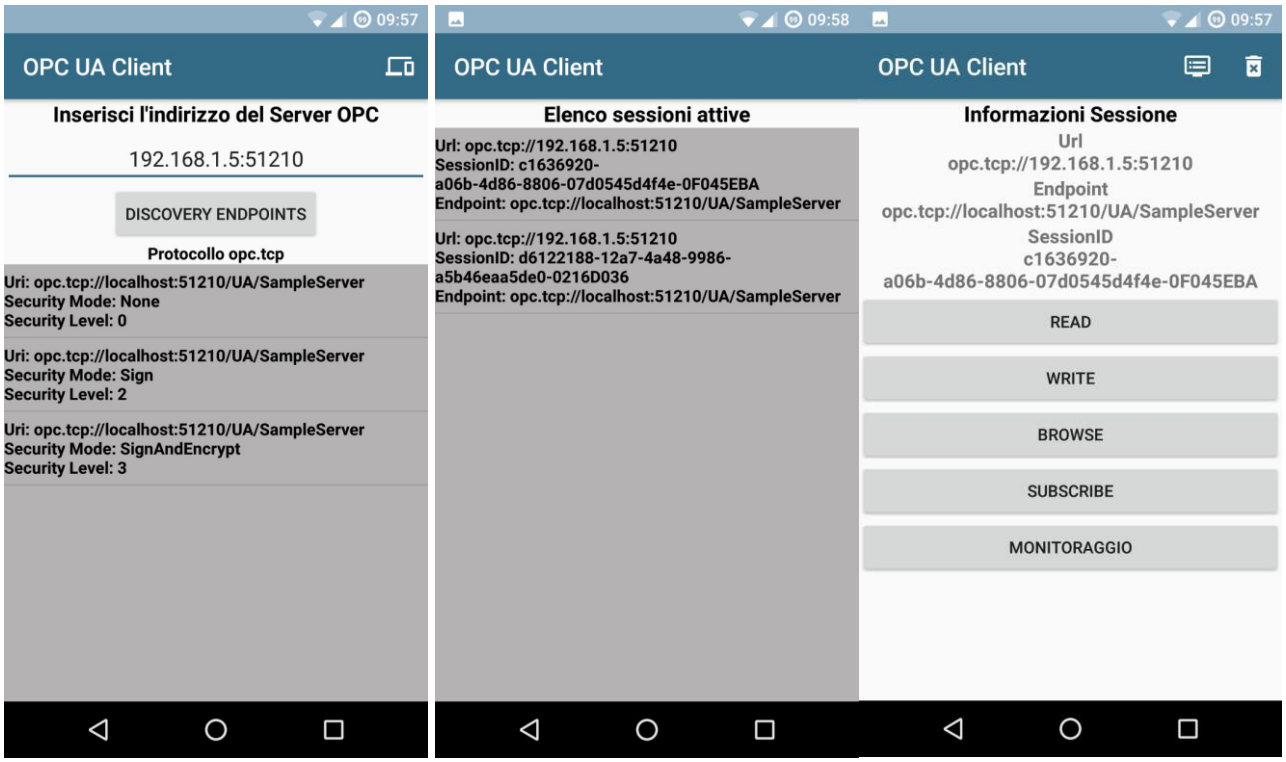


Nel package *OpcUtils* sono state definite tutte le classi di supporto per l'utilizzo e la gestione delle operazioni legate allo stack OPC. In particolare, la classe **GestoreOPC**, implementata come singleton, ha avuto il ruolo di unico punto d'accesso allo stack da tutte le activity di cui è composta l'applicazione. Inoltre, in questo package è stato inserita una sottocartella *ThreadConnessione* che contiene tutte le classi relative ai Thread implementati al fine di eseguire le connessioni al server in quanto su Android, dalla versione 3.0 in poi non è più possibile eseguire chiamate internet dal main thread in modo da rendere più responsiva l'applicazione.

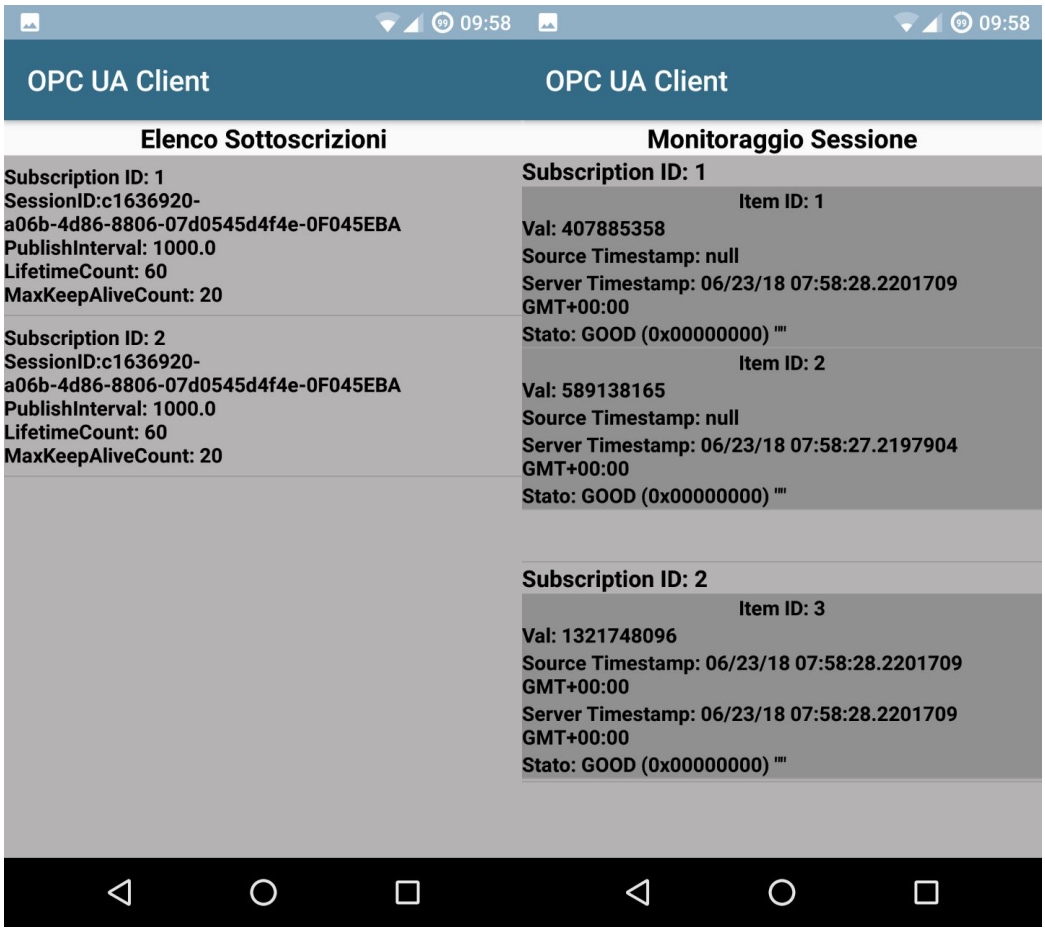
Nel package *org.tinellapreviti.opcclient* sono state inserite tutte le Activity, che servono a gestire le interfacce grafiche di ogni applicazione Android. Infine, nel package *tool.grafici* sono stati inseriti gli Adapter che hanno il compito di consentire la gestione delle numerose liste mostrate dall'applicazione.

Vediamo adesso il dettaglio delle funzionalità dell'applicazione.

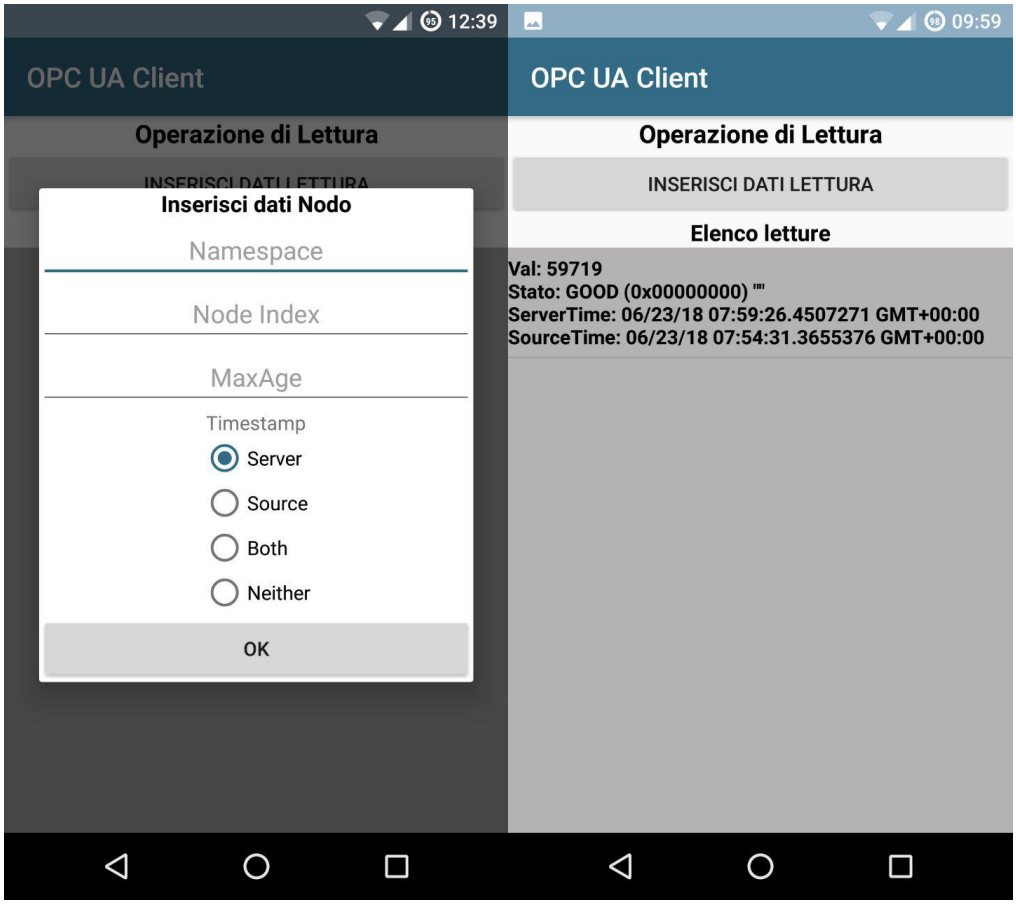
La prima schermata (a sinistra) che si presenta all'utente permette di effettuare il *Discovery* degli Endpoints del server OPC mediante l'inserimento dell'indirizzo IP e la relativa porta. Visualizzati gli endpoints, è possibile tramite la selezione di essi andare ad instaurare una sessione. Fatto ciò verrà visualizzata un'altra schermata (a destra) dalla quale è possibile gestire la sessione. Dalla prima interfaccia è anche possibile accedere (dall'icona presente nella barra in alto) ad una schermata (centrale) dove vengono elencate tutte le sessioni già attivate con il server, da tale elenco è possibile ritornare alla schermata di gestione della sessione specifica.



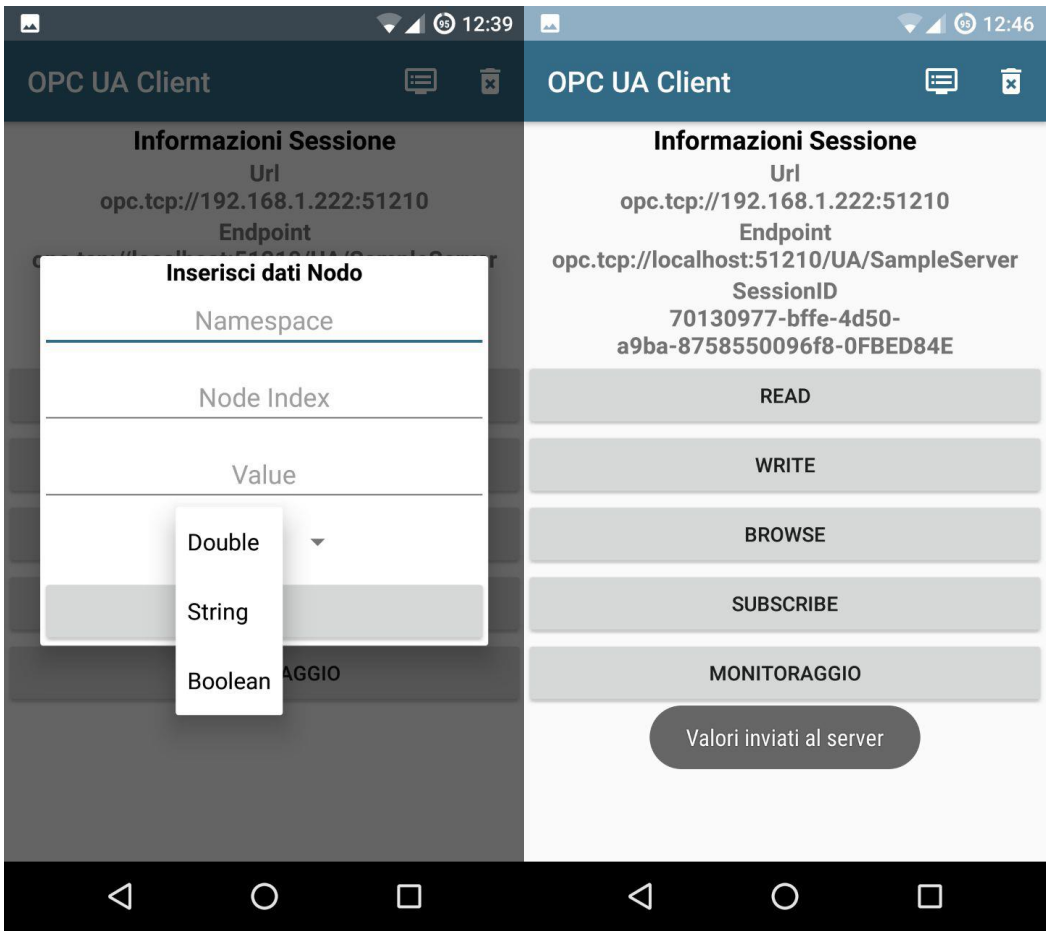
Nella schermata di gestione sessione vengono visualizzate le informazioni caratteristiche della sessione instaurata. Inoltre, viene reso possibile accedere ad altre funzionalità come quelle tipiche messe a disposizione dallo stack come *Read*, *Write*, *Browse* e *Subscribe*, ma anche altre come quella di poter visualizzare un elenco o effettuare un monitoraggio complessivo delle sottoscrizioni precedentemente create. Vi è anche la possibilità di terminare la sessione con il server.



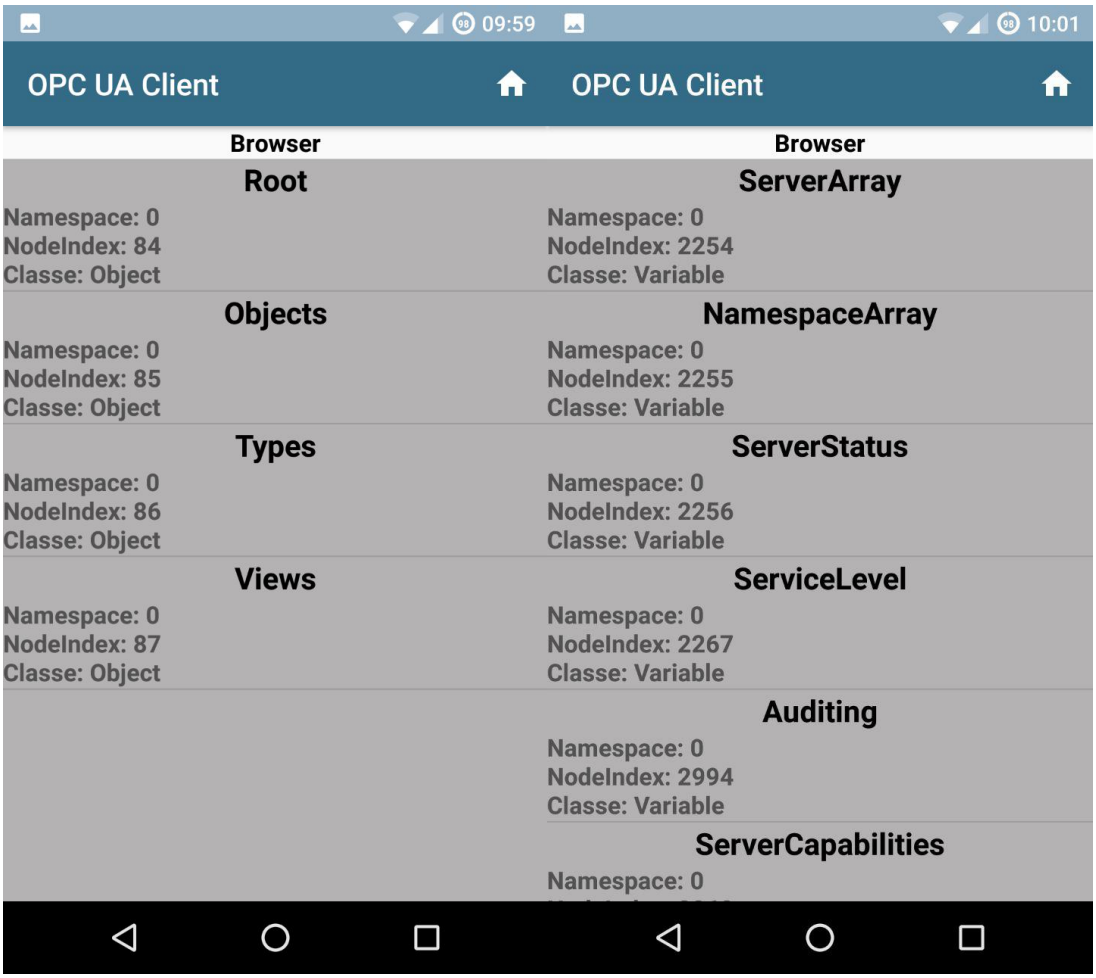
La funzionalità Read è fornita tramite un'interfaccia di dialogo dalla quale è possibile inserire i dati relativi alla lettura come *Namespace*, *NodeIndex*, *MaxAge* e *TimestampType*. I risultati, come il *valore*, lo *stato* e i due *timestamp* sia della sorgente che del server vengono poi elencati nella stessa schermata.



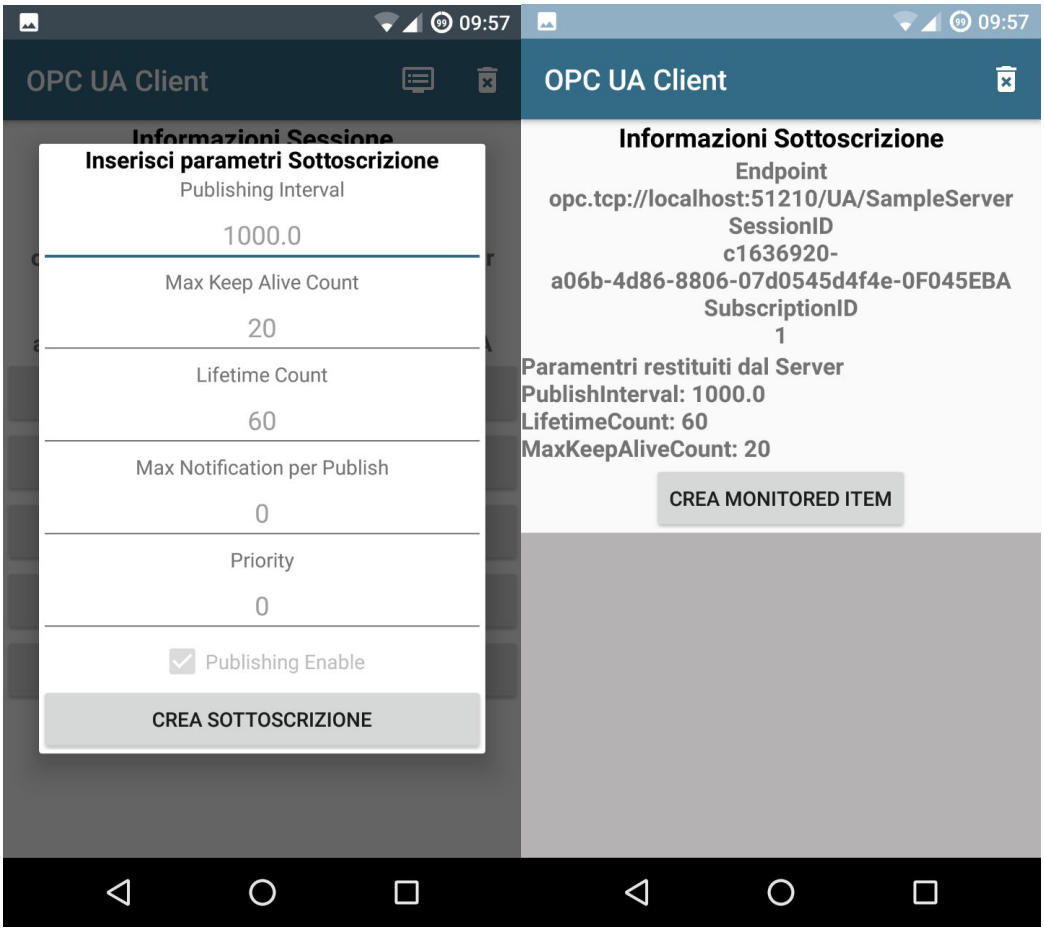
La funzionalità di Write è stata implementata tramite una semplice interfaccia di dialogo dove vengono richiesti i dati relativi alla scrittura ossia *Namespace*, *NodeIndex*, *Value* e tipo del valore (Double, String, Boolean). L'esito dell'esecuzione dell'operazione viene poi visualizzato temporaneamente a schermo tramite un messaggio in sovraimpressione.



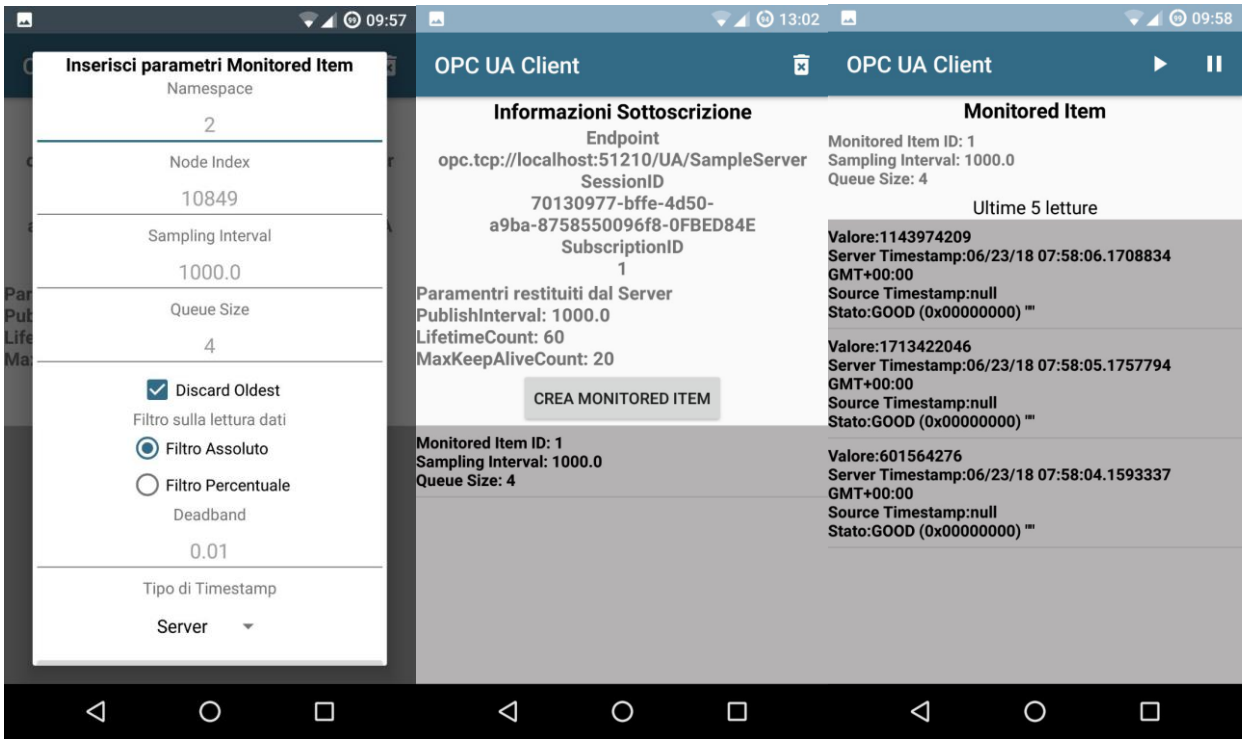
La funzionalità di Browse è stata implementata tramite un elenco selezionabile nel quale vengono visualizzate informazioni relative ai nodi dell'information model a partire dalla Root. Tramite la selezione di ogni nodo, viene aggiornato l'elenco con l'insieme dei nodi accessibili da quello selezionato. Tramite la pressione del tasto indietro l'elenco viene riaggiornato con la precedente visualizzazione. È presente poi nella barra in alto un tasto Home per ritornare direttamente nella schermata di gestione sessione. Per implementare la visualizzazione di tali elenchi è stato necessario l'utilizzo dei Fragment messi a disposizione da Android.



Per la creazione delle sottoscrizioni è stata messa a disposizione una nuova interfaccia di dialogo dalla quale è possibile inserire tutti i parametri caratterizzanti la richiesta di creazione della sottoscrizione. Creata la sottoscrizione, viene mostrata una nuova schermata dalla quale è possibile visualizzare le informazioni concordate con il server riguardo i parametri decisi.



Da questa schermata viene messa a disposizione la funzionalità per la creazione dei Monitored Item tramite una finestra di dialogo dove vengono richiesti i parametri caratterizzanti il singolo item. Una volta che il server conferma la creazione dell'item, vengono aggiunte delle informazioni relative a quest'ultimo in un elenco presente nella schermata della sottoscrizione. Tali elementi dell'elenco sono selezionabili, tramite la selezione si viene portati in un'altra schermata specifica per il singolo MonitoredItem del quale è possibile leggere gli ultimi messaggi ricevuti presenti ancora nel buffer.



CENNI SUL CODICE

GestoreOPC

Parte fondamentale di tutta l'applicazione, la classe singleton GestoreOPC ha il ruolo di essere l'unico mantenitore di punto di accesso per la comunicazione verso il Server OPC.

Nel costruttore viene creata un'istanza della classe Client messa a disposizione dallo stack che verrà poi riutilizzata per la creazione delle sessioni. Vengono inizializzati i nodi base per l'operazione di browse insieme ad una struttura a pila che tornerà successivamente utile per le operazioni di browse. Viene inizializzato anche un array dinamico dove verranno conservate tutte le sessioni create durante l'utilizzo dell'app.

```
private GestoreOPC(){
    nodibase= new ArrayList<>();
    nodibase.add(Identifiers.RootFolder);
    nodibase.add(Identifiers.ObjectsFolder);
    nodibase.add(Identifiers.ViewsFolder);
    nodibase.add(Identifiers.TypesFolder);
    nodibase.add(Identifiers.Server);

    initStack();

    sessions= new ArrayList<>();
}
```

```
public static GestoreOPC CreateGestoreOPC(final File certFile, final File privKeyFile){
        Thread t=new Thread((Runnable) () -> {
                myClientApplication= new Application();

                try {
                    Cert myCertificate = Cert.load(certFile);
                    PrivKey myPrivateKey = PrivKey.load(privKeyFile, Key);
                    keys=new KeyPair(myCertificate,myPrivateKey);
                }catch (Exception e1) {
                    e1.printStackTrace();
                    try {
                        keys = CertificateUtils
                                .createApplicationInstanceCertificate( commonName: "Android_OPC_Client", organisation: "tinellaprevitideveloper",
                                        applicationUri: "org.tinellapreviti.opclient", validityTime: 3650);
                        keys.getCertificate().save(certFile);
                        keys.getPrivateKey().save(privKeyFile, Key);
                    } catch (GeneralSecurityException e) {
                        e.printStackTrace();
                        keys=null;
                    } catch (IOException e) {
                        e.printStackTrace();
                        keys=null;
                    }
                }
                myClientApplication.addApplicationInstanceCertificate(keys);
                myClientApplication.setApplicationUri("org.tinellapreviti.opclient");
                client=  new Client(myClientApplication);
        });
    t.start();
```

La Browse, funzionalità complessa, è stata implementata considerando la gerarchia dei nodi come una Pila. Così facendo si ha modo di lavorare sempre sull'ultimo elemento, rimuovendolo dalla pila durante la risalita della gerarchia in quanto non più necessario. Tale struttura ha permesso di semplificare le operazioni.

```
public BrowseResponse Browse(int position,int session_position) throws ServiceResultException {
    BrowseDescription browse = new BrowseDescription();
    browse.setNodeId(GestoreOPC.getIstance().getNodo(position));
    browse.setBrowseDirection(BrowseDirection.Forward);
    browse.setIncludeSubtypes(true);
    browse.setNodeClassMask(NodeClass.Object, NodeClass.Variable);
    browse.setResultMask(BrowseResultMask.All);

    BrowseResponse res= sessions.get(session_position).getSession()
            .Browse( RequestHeader: null,  View: null,  RequestedMaxReferencesPerNode: null, browse);
    ArrayList<NodeId> nodes= new ArrayList<>();

    for(int i=0;i<res.getResults().length;i++){
        for(int j=0;j<res.getResults()[i].getReferences().length;j++){
            int namespace= res.getResults()[i].getReferences()[j].getNodeId().getNamespaceIndex();

            NodeId node;
            if(res.getResults()[i].getReferences()[j].getNodeId().getValue() instanceof String)
                node= new NodeId(namespace,res.getResults()[i].getReferences()[j].getNodeId()
                        .getValue().toString());
            else
                node= new NodeId(namespace, (UnsignedInteger)res.getResults()[i].getReferences()[j]
                        .getNodeId().getValue());

            nodes.add(node);
        }
    }

    if(nodes.size()>0)
        stack.push(nodes);
    return res;
}
```

Altri metodi importanti che compongono la classe GestoreOPC. In particolare, il metodo CreateSession si occupa di eseguire le operazioni dello stack necessarie per la creazione e attivazione del canale. Esso verrà richiamato all'interno del ThreadCreateSession.

```
public int CreateSession(String url, EndpointDescription endpoint) throws ServiceResultException {
    SessionChannel tmp=client.createSessionChannel(url, endpoint);
    tmp.activate();
    sessions.add(new SessionElement(tmp,url));
    return sessions.size()-1; //posizione in cui è stata aggiunta la nuova sessione
}

public List<SessionElement> getSessions() { return sessions; }

public Client getClient() { return client; }

public NodeId getNodo(int pos) { return stack.peek().get(pos); }

public void initStack(){
    stack= new Stack<>();
    stack.add(nodibase);
}
public void pop(){
    if(stack.size()>1)
        stack.pop();
}
```

La classe *SessionElement*, è un wrapper avanzato che viene utilizzato per organizzare i dati relativi ad ogni singola Sessione, offrendo la funzionalità di creare le sottoscrizioni.

```
public class SessionElement{
    private SessionChannel session;
    private String url;
    private List<SubscriptionElement> subscriptions;

    boolean running=false;
    ThreadPublish thread=null;

    public SessionElement(SessionChannel session, String url) {
        this.session = session;
        this.url = url;
        subscriptions=new ArrayList<>();
    }

    public int CreateSubscription(CreateSubscriptionRequest request) throws ServiceResultException{...}

    public synchronized void stopRunning() { running=false; }
    public synchronized void startRunning() { running=true; }
    public synchronized boolean isRunning() { return running; }

    public SessionChannel getSession() { return session; }

    public String getUrl() { return url; }

    public List<SubscriptionElement> getSubscriptions() { return subscriptions; }
}
```

```
public int CreateSubscription(CreateSubscriptionRequest request) throws ServiceResultException{
    CreateSubscriptionResponse response = session.CreateSubscription(request);
    if(thread != null) {
        try {
            stopRunning();
            thread.join();
            subscriptions.add(new SubscriptionElement(response,this.session));
            startRunning();
            thread= new ThreadPublish( sessionElement: this);
            thread.start();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }else{
        subscriptions.add(new SubscriptionElement(response,this.session));
        startRunning();
        thread= new ThreadPublish( sessionElement: this);
        thread.start();
    }
    return subscriptions.size()-1;
}
```

In particolare, durante la creazione della prima sottoscrizione viene inizializzato e avviato un ThreadPublish(di cui si tratterà più avanti) relativo alla sessione che è unico per tutte le sottoscrizioni ad essa legate.

Anche la classe *SubscriptionElement* ha il ruolo di wrapper avanzato che viene utilizzato per organizzare i dati relativi ad ogni singola sottoscrizione fornendo funzionalità per la creazione dei MonitoredItem.

```
public class SubscriptionElement{
    private CreateSubscriptionResponse subscription;
    private List<MonitoredItemElement> monitoreditems;
    private SessionChannel sessionChannel;

    SubscriptionAcknowledgement subAck;

    public void setLastSeqNumber(UnsignedInteger lastSeqNumber) {
        subAck.setSequenceNumber(lastSeqNumber);
    }

    public SubscriptionAcknowledgement getSubAck() { return subAck; }

    public SubscriptionElement(CreateSubscriptionResponse subscription, SessionChannel sessionChannel) {
        this.subscription = subscription;
        this.monitoreditems= new ArrayList<>();
        this.sessionChannel=sessionChannel;

        subAck = new SubscriptionAcknowledgement();
        subAck.setSubscriptionId(new UnsignedInteger(subscription.getSubscriptionId()));

    }

    public CreateSubscriptionResponse getSubscription() { return subscription; }

    public List<MonitoredItemElement> getMonitoreditems() { return monitoreditems; }

    public SessionChannel getSession() { return sessionChannel; }
```

```
public int CreateMonitoredItem(CreateMonitoredItemsRequest mirequest)
        throws ServiceResultException, MonItemNotCreatedException {
    CreateMonitoredItemsResponse response= sessionChannel.CreateMonitoredItems(mirequest);
    if(response.getResults()[0].getStatusCode().getValue().intValue()!= StatusCode.GOOD.getValue()
            .intValue()) {
        throw new MonItemNotCreatedException(response.getResults()[0].getStatusCode()
                .getDescription());
    }
    monitoreditems.add(new MonitoredItemElement(response,mirequest));
    return monitoreditems.size()-1;
}
```

La classe *MonitoredItemElement* è l'ultima classe wrapper avanzato che viene utilizzato per organizzare i dati relativi ad ogni MonitoredItem.

```
public class MonitoredItemElement {
    private CreateMonitoredItemsResponse monitoreditem;
    private CreateMonitoredItemsRequest monitoreditem_request;
    private LinkedList<MonitoredItemNotification> letture;
    public static final int buffersize=5;

    public MonitoredItemElement(CreateMonitoredItemsResponse monitoreditem,
                                CreateMonitoredItemsRequest monitoreditem_request) {
        this.monitoreditem = monitoreditem;
        this.monitoreditem_request=monitoreditem_request;
        letture= new LinkedList<>();
    }

    public CreateMonitoredItemsResponse getMonitoreditem() { return monitoreditem; }

    public LinkedList<MonitoredItemNotification> getLetture() { return letture; }

    public void insertNotification(MonitoredItemNotification notification){
        if(letture.size()==buffersize)
            letture.removeLast();
        letture.addFirst(notification);
    }

    public CreateMonitoredItemsRequest getMonitoreditem_request() { return monitoreditem_request; }
}
```

La classe *ThreadPublish*, di cui vengono create tante istanze per quante sono le sessioni attive, ha il ruolo di andare a prelevare tutti i messaggi prodotti dai monitoredItem. Esso avrà un riferimento alla sessione che lo ha richiamato in modo tale da poter accedere a tutte le informazioni e strutture richieste.

```
public class ThreadPublish extends Thread {

    SessionElement sessionElement;
    public ThreadPublish(SessionElement sessionElement) {
        this.sessionElement = sessionElement;
    }
```

In particolare, questo è il codice relativo al run del Thread che ha il compito di effettuare la Publish al server forzandolo così a inviare i messaggi che andrà successivamente ad inserire nei buffer dei relativi monitoredItem.

```
while(sessionElement.isRunning()){
    if(sessionElement.getSubscriptions().size()>0){
        PublishResponse publishResponse;
        try {
            SubscriptionAcknowledgement[] subacks= new SubscriptionAcknowledgement[sessionElement.getSubscriptions().size()];
            for(int i=0;i<sessionElement.getSubscriptions().size();i++){
                subacks[i]=sessionElement.getSubscriptions().get(i).getSubAck();
            }
            publishResponse = sessionElement.getSession().Publish( RequestHeader: null, subacks);
            for(int i=0;i<sessionElement.getSubscriptions().size();i++){
                if(sessionElement.getSubscriptions().get(i).getSubscription().getSubscriptionId().getValue()==
                        publishResponse.getSubscriptionId().getValue()){
                    sessionElement.getSubscriptions().get(i).setLastSeqNumber(publishResponse
                            .getNotificationMessage().getSequenceNumber());
                    NotificationMessage nm = publishResponse.getNotificationMessage();
                    ExtensionObject[] ex = nm.getNotificationData();
                    for (ExtensionObject ob : ex) {
                        Object change = ob.decode(GestoreOPC.getIstance().getClient().getEncoderContext());
                        if (change instanceof DataChangeNotification) {
                            DataChangeNotification dataChange = (DataChangeNotification) change;
                            MonitoredItemNotification[] mnchange = dataChange.getMonitoredItems();
                            for (MonitoredItemNotification monitoredItemNotification : mnchange) {
                                for (int j = 0; j < sessionElement.getSubscriptions().get(i).getMonitoreditems().size(); j++) {
                                    if (monitoredItemNotification.getClientHandle().intValue() ==
                                            sessionElement.getSubscriptions().get(i).getMonitoreditems()
                                    .get(j).getMonitoreditem_request().getItemsToCreate()[0]
                                    .getRequestedParameters().getClientHandle().intValue()) {
                                    sessionElement.getSubscriptions().get(i).getMonitoreditems()
                                            .get(j).insertNotification(monitoredItemNotification);
                                    break;
                                    }
                                }
                            }
                        }
                    }
                    break;
                }
            }
        } catch (ServiceResultException e) {}
        try {
            Thread.sleep( millis: 100);
```

Questa sezione di codice tratta dalla classe *ThreadRead,* ha lo scopo di effettuare una chiamata al server eseguendo una Read. Le altre classi riguardanti i Thread(esclusa ThreadPublish), hanno una struttura simile.

```
@Override
public void run() {
    super.run();
    try {
        Thread t= new Thread((Runnable) () → {
                try {
                    ReadResponse res;
                    if(nodeId_String==null)
                        res = session.Read( RequestHeader: null, MaxAge, timestamp,
                                new ReadValueId(new NodeId(namespace, nodeId), attribute, IndexRange: null,  DataEncoding: null));
                    else
                        res = session.Read( RequestHeader: null, MaxAge, timestamp,
                                new ReadValueId(new NodeId(namespace, nodeId_String), attribute, IndexRange: null,  DataEncoding: null));

                    send(handler.obtainMessage( what: 0,res));
                } catch (ServiceResultException e) {
                    send(handler.obtainMessage( what: -1));
                }
        });
        t.start();
        t.join( millis: 8000);
        send(handler.obtainMessage( what: -2));
    } catch (InterruptedException e) {
        send(handler.obtainMessage( what: -2));

    }

}
```

Questa sezione di codice è stata tratta dalla classe *ReadActivity* e si occupa di istanziare un Thread per l'esecuzione dell'operazione Read, creando anche un *Handler* che ha il ruolo di callback che servirà per aggiornare l'interfaccia con i risultati o eventualmente comunicare messaggi di fallimento. Anche le altre activity hanno una struttura simile durante la richiamata delle operazioni dello stack OPC. Si differiscono principalmente per la gestione delle interfacce grafiche e interazione con l'utente.

```
ThreadRead t;
if(nodeid_string==null)
    t = new ThreadRead(sessionElement.getSession(), maxAge, timestamps, namespace, nodeid, Attributes.Value);
else
    t = new ThreadRead(sessionElement.getSession(), maxAge, timestamps, namespace, nodeid_string, Attributes.Value);

progressDialog = ProgressDialog.show( context: ReadActivity.this,  title: "Tentativo di connessione",  message: "Lettura in corso...",  indeterminate: true);
Handler handler = handleMessage(msg) → {
        if (msg.what == -1) {
            Toast.makeText(getApplicationContext(),  text: "Non è stato possibile effettuare la lettura", Toast.LENGTH_LONG).show();
        } else if (msg.what == -2) {
            Toast.makeText(getApplicationContext(),  text: "Il server non risponde", Toast.LENGTH_LONG).show();
        } else {
            ReadResponse res = (ReadResponse) msg.obj;
            listaLetture.add(res);
            adapter.notifyDataSetChanged();
            listRead.setSelection(adapter.getCount() - 1);
            progressDialog.dismiss();
        }
};
t.start(handler);
dialog.dismiss();
```

La classe *MonitoraggioAdapter*, come gli altri Adapter, ha il ruolo di gestire i singoli elementi all'interno delle liste contenenti le informazioni che vengono visualizzate all'utente.

```
public class MonitoraggioAdapter extends ArrayAdapter<SubscriptionElement> {

    SubMonitoraggioAdapter adapter;

    public MonitoraggioAdapter(Context context, int resource, List<SubscriptionElement> objects) {
        super(context, resource, objects);
    }

    @Override
    public View getView(int position, View convertView, ViewGroup parent) {
        LayoutInflater inflater= (LayoutInflater) getContext().getSystemService(Context.LAYOUT_INFLATER_SERVICE);
        convertView= inflater.inflate(R.layout.list_monitoraggio, root: null);
        SubscriptionElement obj= getItem(position);

        TextView txtSubID= convertView.findViewById(R.id.txtSubID);
        ListView listSub= convertView.findViewById(R.id.listSubMonitored);

        ViewGroup.LayoutParams l= listSub.getLayoutParams();
        l.height= (int)130dp*obj.getMonitoreditems().size();
        listSub.setLayoutParams(l);

        txtSubID.setText("Subscription ID: "+obj.getSubscription().getSubscriptionId());
        adapter= new SubMonitoraggioAdapter(getContext(),R.layout.list_submonitoraggio,obj.getMonitoreditems());

        listSub.setAdapter(adapter);

        return convertView;
    }
}
```